CGA INC.
D2 PROJECT PROPOSAL

TELEGRAM

SWEN 3145

Chazana Johnson: 620121088
Glenroy Logan: 620099685
Amy Daharu : 816015107

Table Of Contents

Table of Figures

List Of Changes

Original Assignment	Changes/Corrections Since Original Assignment was submitted
D1 -	Pointers were given and the necessary adjustments were made to the description of the system to be modelled. The description now captures the system to be modelled as a whole, with all the features such as a user of the system being able to have multiple accounts and to access these accounts from numerous devices being represented throughout the design. The design patterns mentioned earlier in the description was also advised on and has since been updated and incorporated to produce a model reflecting them.
D2 - Weaknesses identified by group members and the lecturer included weak modelling skills, the class diagram presented required some restructuring to better represent the system. It was advised to familiarize ourselves with the application more before moving forward. Object diagrams were deemed incomplete and advice was given as to how to produce a complete object diagram.	• The addition of an attribute to the manageAccount class named nbrofUserProfiles, which takes in an integer, this attribute accounts for the amount of user profiles which a user possesses which can range from one to three accounts. • The addition of an association (Profile) between the manageAccount and the User class, this accounts for one user having the ability to own up to 3 accounts. • The addition of an association (accountProfileSettings) the manageAccount class with the UpdateProfileSettings, this will allow a user to update the user profile for each of their accounts individually. • Attributes within the UpdateProfileSettings class were updated to profilePicture which uses the enumeration Media Type and characterCount which accepts an Integer. • Attributes galleryAccessPermission and cameraAccessPermission were added to the device class, this accounts for the user allowing or denying permission for the application to access their device gallery and to use their camera. • Attributes userName and status were added to the User class, this accounts for the userName associated with the user's profile as well as the status of the user whether they are online or offline. • The Contact class name was updated to ContactList and the attributes contactCount, username and validNumber were added to the class. • An enumeration callStatus was added to the class diagram and the attribute status was added to the call class which uses this enumeration to reflect the current status of a class, whether it be Ringing, Connected, Disconnected, Unanswered or HangupCall. • An attribute groupMemberAdmins which accounts for the amount of group members which are also admins was added to the Administrator class. • The attributes characterCount and groupBots were added to the Group class and the attribute UserID was removed from said class.

	• The channel class which inherits the Group class was modeled into the class diagram and an association (channelMessage) was made between the channel and Message class. • The sender and receiver class which was associated with the Message class was removed and a new class(chat) was added which inherits the Message class and forms an association chatMessage with the Message class. • SecretChat was removed from within the MediaType enumeration and a class called SecretChat was created, this class inherits the Message class and forms an association (SecretChatMessage) with the User class. The following associations were formed between the secret chat and call class, secretAudioCall and secretVideoCall. Another association secretContact was also formed between the contactLit and secret chat class. • The association (ForwardMessage) formed between the Message class which was associated with itself was removed. • The association(contactList) formed between the user class which associated with itself was removed. • The association (UserCall) which was formed between the Call and User class was removed and replaced with two new associations between the two classes named AudioCall and VideoCal.
D3 - Weakness in modelling was present in this document. It was advised to restructure the class model presented to reflect the description which stated the design patterns to be used in the system. Pointers were also given as to how to represent the design patterns mentioned in the class diagram.	The entire class diagram was remodeled based on feedback given as well as based on reflection done by the group. The structural changes made from D3 to D4 includes: • A class called Telegram was introduced, this class allows a user to create, reactivate and deactivate an account on the application. • The Profile class was added and is now the main class. It is from this class where most major functions are carried out such as sending a message and editing your profile picture, Biography, username or changing your phone number. • The User class was reduced and now is limited to allowing users to add or delete a profile. • The UpdateSettings class was removed. The attributes which were within this class are now operations and can now be found within the Profile class list of operations. • The ManageAccount class was also removed. • The contact list is now linked to the Profile class as a reflexive association. • A delete profile operation has been added to the user class allowing the user to delete one or more of their userprofile. • The device class now holds the attributes for permission access to users gallery, camera and microphone. Additional changes include the renaming and redefinition of multiplicities within the class diagram.

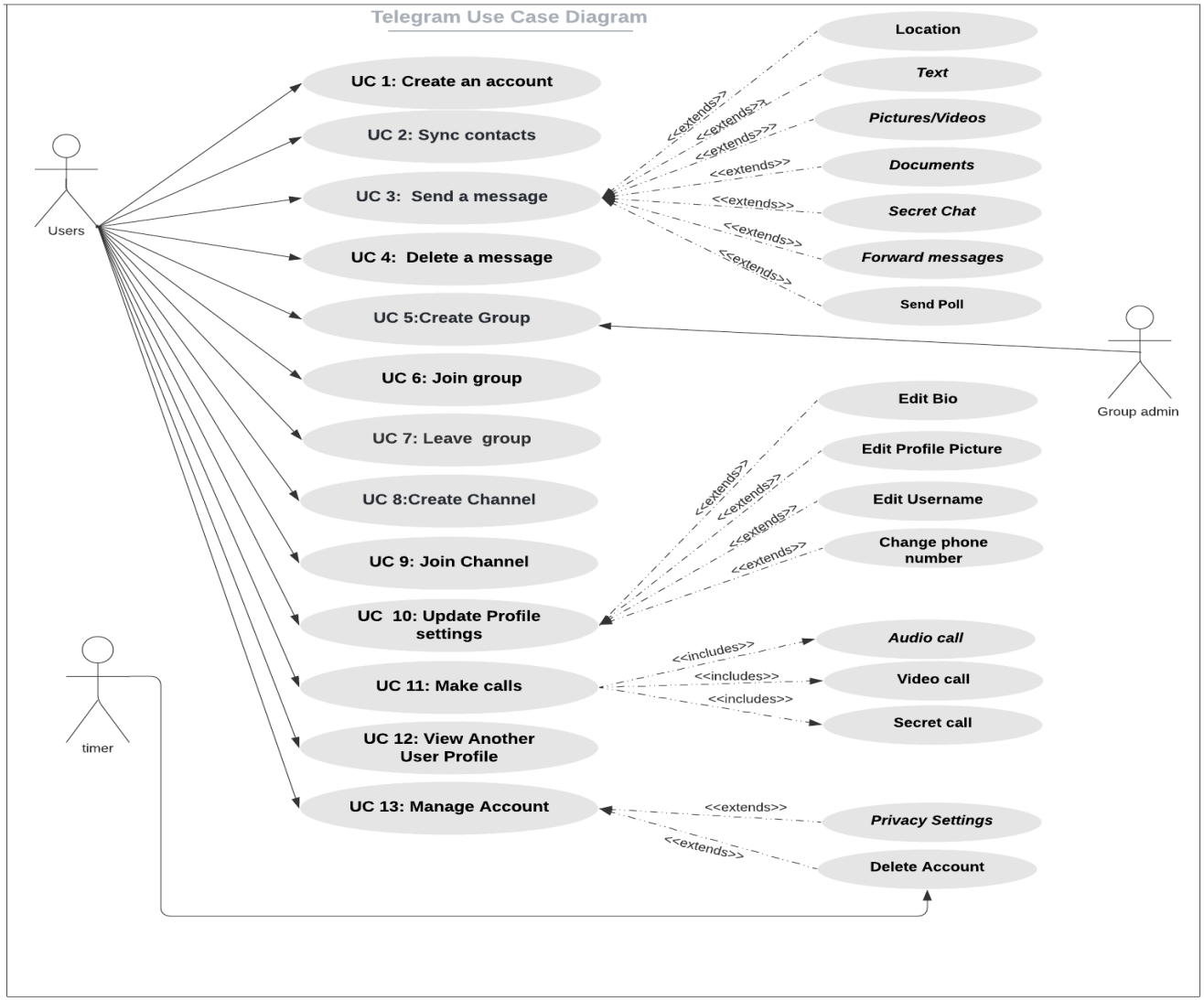
Project Description

Telegram is an open-source, cloud-based instant messaging application established in 2013 and has been growing in popularity since, it has over 500 million active users and was the most downloaded app in January of 2021. Telegram can be used on smartphones, tablets, laptops and desktop computers and is available for Android, iOS, Windows Phone, Windows NT, macOS and Linux. Telegram allows its users to send/receive text, images, audio and video messages to other users of the application which are a part of the users synchronized contact list within the application, users of the application can send/receive files up to 2GB in size. Telegram's distinctive feature is security and it provides client-to-server encryption (on regular chats) as well as end-to-end encryption (on secret chats) on text, audio and video messages being transferred as well as end-to-end encryption during audio and video calls. There's also the option to test the security of your 'Secret Chats' by using an image as an encryption key. You can effectively verify that your chat is secure and less prone to man-in-the-middle attacks by comparing your encryption key to a friend's. Though the application is cloud based, it also relies on its users' device storage as all data downloaded or sent via Telegram is stored in the cache of the users' device. Users can free up device storage by clearing Telegram cache storage; this deletes the stored data on the user's device. Though removed from the device users' information will still be in the cloud and can be redownloaded if needed, Telegram also enables its users to set self-destruct timers on secret chat messages and on their accounts if they have been inactive for a period of time.

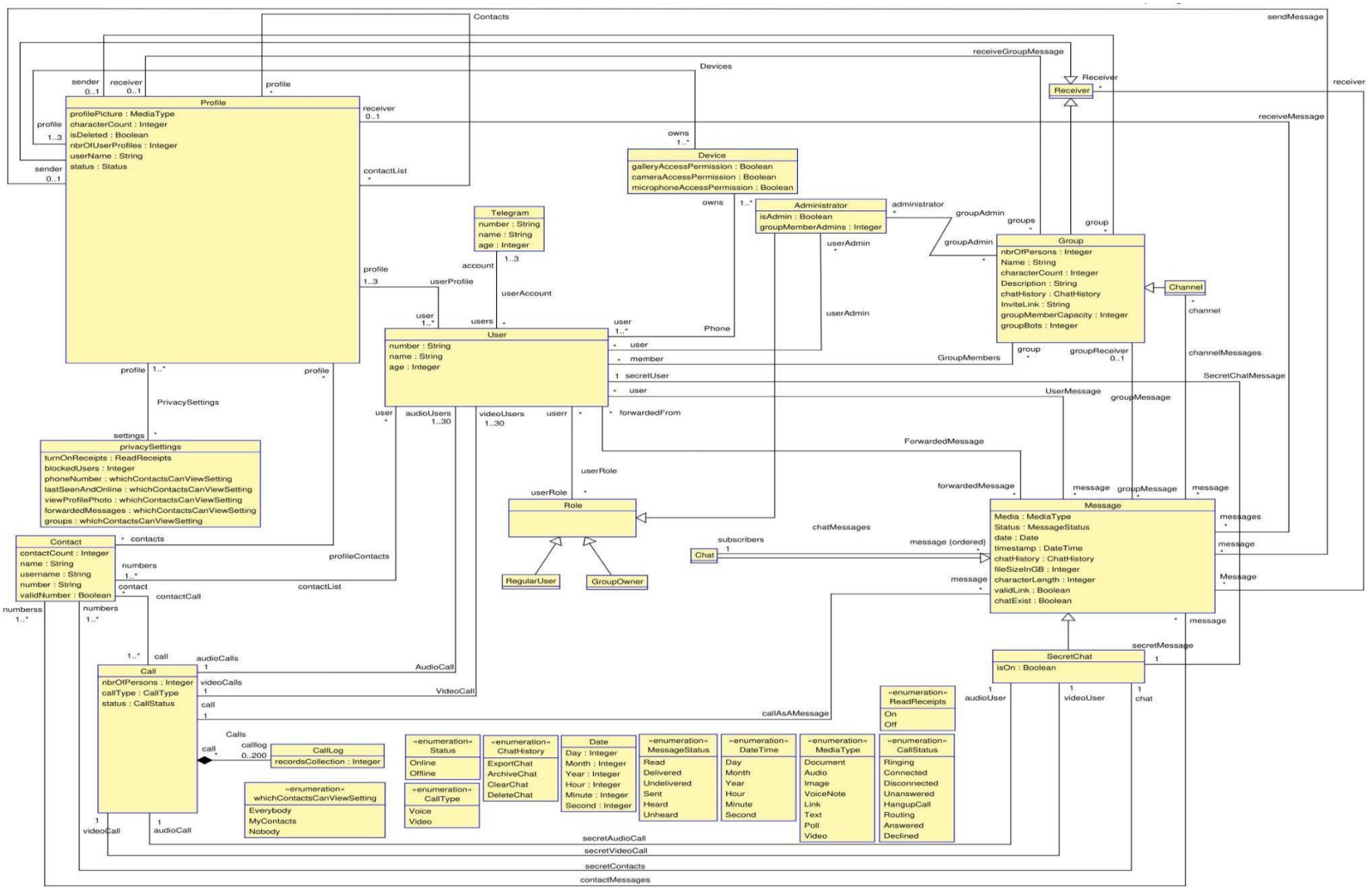
Telegram allows its users to do a vast array of things such as:

- Create usernames which users which they are a part of their contact list can find them in the application by.
- Creating a channel – A Telegram channel is one way messaging where admins are able to post messages but other users are not, any user is able to create and subscribe to channels, channels can be created for broadcasting messages to an unlimited number of subscribers. (Channels are public and has an unlimited number of subscribers)
- Create group chats with up to 200,000 members.
- Forward messages you sent/received to members of your Telegram contact list.
- Set notifications to be repeated after a period of time if they were not read upon receipt.
- Search a chat using a number/letter/key words or phrases.
- Clear chat history.
- Mute chat notifications for an hour, 8 hours, 2 days or none at all.
- Block contact members from messaging you via Telegram.
- Synchronize their phone contact list to Telegram, making it easier for them to find and message them via their saved contact name in Telegram.

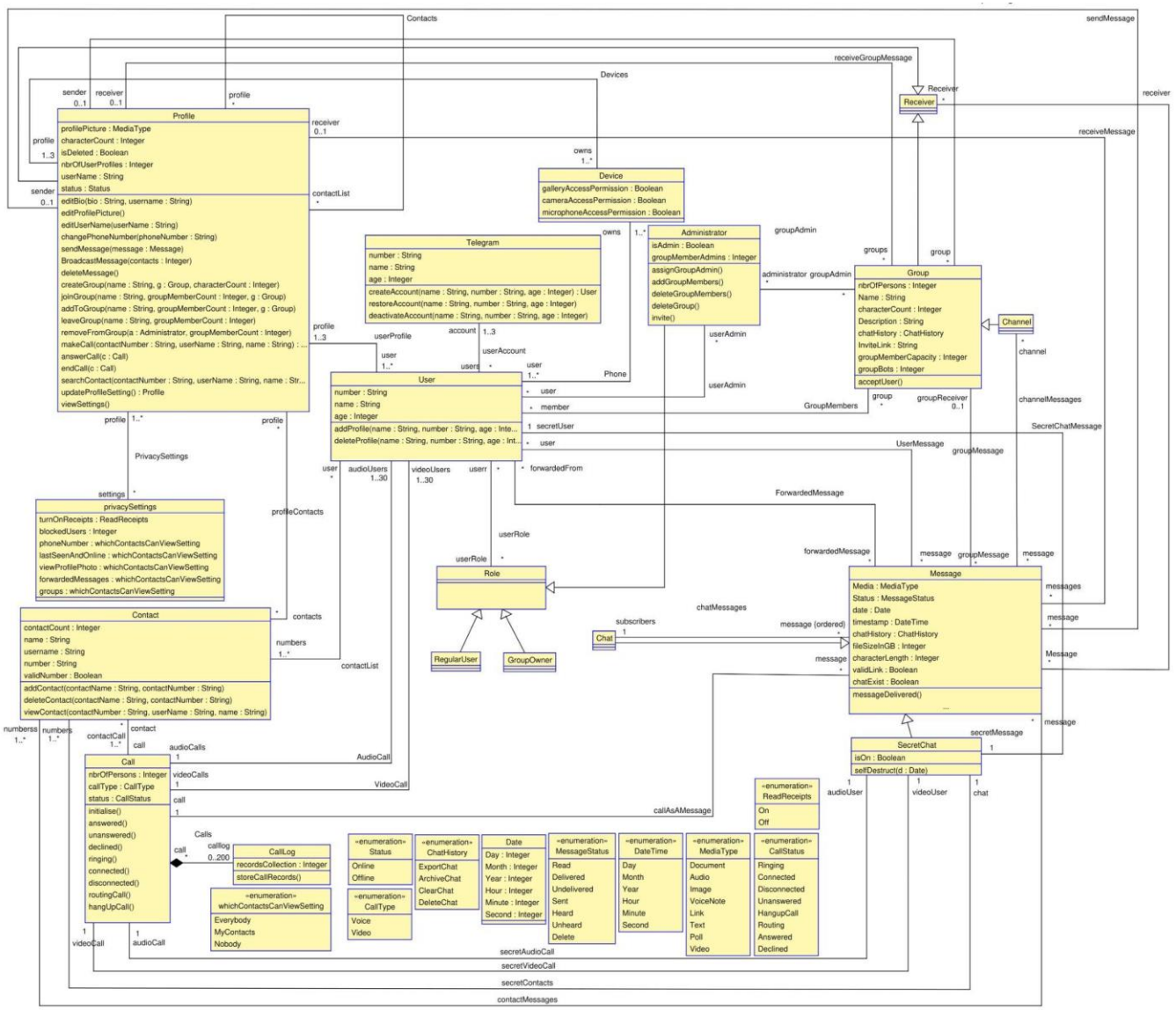
Use Case Diagram



Requirements Level Class Diagram



Design Level Class Diagram



Use Case Classification

Table showing all use cases and classes in the model and primary responsibilities for each use case

Table of Use Cases - Classes

Use Case	Class with Primary Responsibility
1.Create an Account	Telegram
2.Sync contacts	User
3.Send a message	Profile
4. Delete a message	Profile
5. Create Group	Profile
6.Join Group	Profile
7.Leave Group	Profile
8. Create Channel	Profile
9. Join Channel	Profile
10. Update Profile Settings	UpdateProfileSettings
11. Make Calls	Profile

12. View Another User Profile	Profile
13. Manage Account	Profile

Table showing the three operations with their OCL pre and postcondition constraints

Name	**English Description**	**OCL Version**	**Dependent Operations**
addContact(contactName: String, contactNumber: String) ***Taken from the Contact Class***	Adding a contact to the user's contact list. contactNameIsNotNull: Contact Name cannot be set to null contactNumberDoesNotExist: The contact number does not already exist contactNumberIsUnique: The contact number is unique isValidPhoneNumber: The phone number is valid contactListToBeUpdated: The Contact List to be updated after operation contactAdded: The contact is now added	**addContact**(contactName: String, contactNumber: String) pre **contactNameIsNotNull**: self.user -> select(c|c.name = contactName <> null) -> size() = 0 pre **contactNumberDoesNotExist**: self.user->select(n|n.number = contactNumber) -> size() = 0 pre **contactNumberIsUnique**: self.user->select(n|n.number = contactNumber) -> isUnique(contactNumber) pre **isValidPhoneNumber**: self.validNumber = true post **contactListToBeUpdated** : self.user -> size() = self.user@pre->size()-1 post **contactAdded**: self.user->select (n| n.name = contactName and n.number = contactNumber) -> size() = 1	null

deleteContact*(*contactName: String, contactNumber: String*)* ***Taken from the Contact Class***	Removing a contact from the user's contact list. contactExists:That the contact Exists within the contact List contactDeleted:The contact is deleted	**deleteContac**t(contactName: String, contactNumber: String) pre **contactExists**: self.user -> select (n| n.name = contactName and n.number = contactNumber) -> size() = 1 post **contactDeleted**: let u = self.user -> select (n| n.name = contactName and n.number = contactNumber) -> size() - 1 in user@pre = user->excluding(u)	addContact(contactName: String, contactNumber: String)

editBio(bioName: String, username: String) ***Taken from the Profile Class***	Allows the user to edit their bio on telegram. bioNameMustBeAdded: The user's bio must be added already if we are editing it. maxBioCharacterCount: The users bio has a maximum of 70 characters post: The Bio name is updated	editBio(bioName: String, username: String) pre **bioNameMustBeAdded**: self.device.profile -> select (n| n.name = bio and self.characterCount < 500) -> size() = 1 pre **maxBioCharacterCount**: self.characterCount <= 70 post: self.device.profile -> select (n| n.name = bio = true and self.characterCount < 500 = true) -> size() = 1	createAccount(Name:String,Number: String,age: Integer)

editProfilePicture()
Taken from the Profile Class
	Allows the user to change their profile picture.
	
	permissionToAccessGalleryOrCamera: The app has to have permission/access to use gallery and camera
	
	profilePictureExists: A profile picture would have had to exist before inorder to update it to a new one
		editProfilePicture()
		
		pre **permissionToAccessGalleryOrCamera**:self.device ->select(access | access.galleryAccessPermission = true or access.cameraAccessPermission = true) -> size() = 1
		
		pre **profilePictureExists**:
		if self.device.user -> notEmpty()
		then self.profilePicture = MediaType::Image = true
		else
		self.profilePicture = MediaType::Image = false
		endif
		
		post **profilePictureIsUpdated**:
			createAccount(Name:String,Number: String)

	profilePictureIsUpdated: The profile picture is added	if self.device.user@pre-> notEmpty()
		then self.profilePicture = MediaType::Image = true and self.device ->select(access | access.galleryAccessPermission = true or access.cameraAccessPermission = true) -> size() = 1
		else
		self.profilePicture = MediaType::Image = false
		endif	

editUserName(userName: String) ***Taken from the Profile Class***	Allows the user to edit their username. minUserNameCharacters: The minimum character count for the username must be 5 or more userNameIsUnique: The user name must be unique userNameIsAdded: The username is added	editUserName(userName: String) pre **minUserNameCharacters**: self.device.user -> select (n| n.name = userName and self.characterCount > 5) -> size() = 1 pre **userNameIsUnique**: self.device.user -> select (n| n.name = userName) -> isUnique(userName) post **userNameIsAdded**: self.device.user@pre -> select (n| n.name = userName and self.characterCount < 500) -> size() = 1	createAccount(Name:String,Number: String)

changePhoneNumber(phoneNumber: String) ***Taken from the Profile Class***	A user changing their phone number. phoneNumberExists: A previous phone number must exist phoneNumberIsValid: The phone number is valid phoneNumberIsUnique: the - phone number is unique phoneNumberIsUpdated:The new phone number has been added	changePhoneNumber(phoneNumber: String) pre **phoneNumberExists**: self.user -> select (n| n.number = phoneNumber) -> exists(phoneNumber = true) pre **phoneNumberIsValid**: self.user.contacts -> select(v | v.validNumber = true) -> size() = 1 pre **phoneNumberIsUnique**: self.user -> select (n| n.number = phoneNumber) -> isUnique(phoneNumber) post **phoneNumberIsUpdated:** self.user -> select (n| n.number = phoneNumber) -> size() = 1	createAccount(Name:String,Number: String)

sendMessage*(*message: Message*)* ***Taken from the Profile Class***	Users can send messages to other telegram users. accountMustExist:Contacts can be searched for via names in contact list or user name maxMediaFileSize:All MediaTypes excluding Text,Link and Polls limits are a maximum of 2GB	**sendMessage**(message: Message) pre **accountMustExist**: self.contacts -> select(c | c.name = true or c.username = true) -> size() = 1 pre **maxMediaFileSize**: self.user.message -> select(m | m.Media = MediaType::Document or m.Media = MediaType::Audio or m.Media = MediaType::Image or	addContact(contactName: String, contactNumber: String)

	maxTextMessageLength: The maximum text message character length is 4096 characters linkMustBeValid: all Links must be valid confirmMessageWasSent: The messages from user are confirmed to be sent timeStampMessageWasSent :The Time when messages were sent is recorded	m.Media = MediaType::VoiceNote or m.Media = MediaType::Video and m.fileSizeInGB <= 2) -> size() = 1 pre **maxTextMessageLength**: self.user..message -> select(t | t.Media = MediaType::Text and t.characterLength <= 4096) -> size() = 1 pre **linkMustBeValid**: self.user.message -> forAll(m| m.Media = MediaType::Link and message.validLink = true) post **confirmMessageWasSent**: self.user.message@pre -> select(s |s.Status = MessageStatus::Sent)-> size() = 1 post **timeStampMessageWasSent**: self.user.message@pre -> select(t |t.timestamp = DateTime::Day and t.timestamp = DateTime::Month and t.timestamp = DateTime::Year and t.timestamp = DateTime::Hour and t.timestamp = DateTime::Minute and t.timestamp = DateTime::Second) -> size() = 1	
BroadcastMessage(name: String) ***Taken from the Profile Class***	Allows the user to send a message to multiple contacts. accountMustExist:Contacts can be searched for via names in contact list or user name	**BroadcastMessage**(contacts: Integer) pre **accountMustExist**: self.user.numbers -> select(c | c.name = true or c.username = true) -> size() = 1	addContact(contactName: String, contactNumber: String)

	selectContact: at least one contact must be selected to broadcast message
	
	sendMessaageToAllSelectedContacts: Messages of all types that are less than 2GB are sent to all selected contacts
	
	confirmMessageWasSent: The messages from user are confirmed to be sent
		pre **selectContact**:
		if contacts >= 1
		then self.user.numbers -> select(c| c.contactCount = contacts) -> size() = 1
		else
		contacts < 1 and self.user.numbers -> select(c| c.contactCount = contacts) -> size() = 0
		endif
		
		post **sendMessaageToAllSelectedContacts**: self.user.message@pre -> forAll(m | m.Media = MediaType::Document or m.Media = MediaType::Audio or m.Media = MediaType::Image or m.Media = MediaType::VoiceNote or m.Media = MediaType::Video and m.fileSizeInGB <= 2)
		
		post **confirmMessageWasSent**: self.user.message@pre -> select(s |s.Status = MessageStatus::Sent)-> size() = 1	

deleteMessage() ***Taken from the Profile Class***	Allows the user to delete a message. chatMustExist: A chat must exist between users. messageWasSent: A message must be sent messageIsDeleted: The message Is deleted	**deleteMessage()** pre **chatMustExist**: self.user.message -> select(d| d.chatExist = true) -> size() = 1 pre **messageWasSent**: self.user.message -> select(s | s.Status = MessageStatus::Sent) -> size() = 1 post **messageIsDeleted**: let d =	sendMessage*(*message: Message*)*

		self.user.message@pre->select(m | m.Media = MediaType::Document or m.Media = MediaType::Audio or m.Media = MediaType::Image or m.Media = MediaType::VoiceNote or m.Media = MediaType::Video) -> size() = 1 in self.user.message@pre = self.message ->excluding(d) = true	
createGroup (name: String, g: Group, characterCount: Integer) ***Taken from the Profile Class***	A user can create a group containing multiple contacts. accountMustExist: A contact must exist groupNameCharacters: A group name character must be greater than 0 groupCapacity: A group capacity must be between 1 and 200,000 numberOfPersonsInGroup: number of persons in a group must not exceed the group capacity	**createGroup** (name: String, g: Group, characterCount: Integer) pre **accountMustExist**: self.user.contacts -> select(c | c.name = true or c.username = true) -> size() = 1 pre **groupNameCharacters**: self.user.group -> select(n| n.Name = name and n.characterCount > 0) -> size() = 1 pre **groupCapacity**: g.groupMemberCapacity >= 1 and g.groupMemberCapacity <= 200000 pre **numberOfPersonsInGroup**: g.nbrOfPersons < g.groupMemberCapacity post **isNowAdministrator**: self.group.administrator ->	addContact(contactName: String, contactNumber: String) addToGroup(name: String,groupMemberCount: Integer,g: Group) assignGroupAdmin() addGroupMembers() invite()

	isNowAdministrator: The user is now an Administrator for the group they created groupExists: The group is now created	select(i | i.isAdmin = true) -> size() = 1 post **groupExists**: self.group -> select(n |n.Name = name) -> exists(name = true)	

joinGroup(name: String,groupMemberCount: Integer,g:Group)
Taken from the Profile Class
	This allows a user to join a group.
	
	groupExists: The group must exist
	
	addedToGroupViaLinkOrAdmin: A contact must be added to a group via link or be added by an administrator
		joinGroup(name: String,groupMemberCount: Integer,g:Group)
		
		pre **groupExists**: self.user.group -> select(n |n.Name = name) -> exists(name = true)
		
		pre **addedToGroupViaLinkOrAdmin**:
		if self.message -> select(m| m.Media = MediaType::Link and message.validLink = true)--or self.group.administrator -> select(i | i.isAdmin = true) -> size() = 1
		then self.group -> size() = self.group -> select(n |n.nbrOfPersons = groupMemberCount) -> size() = 1
		else
		self.group -> size() = self.group -> select(n |n.nbrOfPersons = groupMemberCount) -> size() = 0
		endif
			createGroup (name: String, g: Group, characterCount: Integer)

	groupCapacity: group capacity must be between 1 and 200,000 addNewGroupMember: A user is now a member of the group	pre **groupCapacity**: g.groupMemberCapacity >= 1 and g.groupMemberCapacity <= 200000 post **addNewGroupMember**: self.user.group -> size() = self.user.group -> select(n |n.nbrOfPersons = groupMemberCount) -> size() = 1	
addToGroup(name: String,groupMemberCount: Integer,g: Group) ***Taken from the Profile Class***	This allows a user to add participants to a group. contactMustExist: A contact must exist (contacts can be added either via contact list or by user name) groupCapacity: A group capacity must be less than 200,000 members addNewGroupMember: A user is now a member of the group	**addToGroup**(name: String,groupMemberCount: Integer,g: Group) pre **contactMustExist**: self.user.numbers -> select(c | c.name = true or c.username = true) -> size() = 1 pre **groupCapacity**: g.groupMemberCapacity <= 200000 post **addNewGroupMember**: self.user.group -> size() = self.user.group -> select(n |n.nbrOfPersons = groupMemberCount) -> size() = 1	createGroup (name: String, g: Group, characterCount: Integer) assignGroupAdmin()

leaveGroup(name: String,groupMemberCount: Integer) ***Taken from the Profile Class***	This allows a user to leave a group themselves. groupExists: A group must exist userMustExistInGroup: A user must be an existing member of the group deleteGroupMember: A user is no longer a member of the group	**leaveGroup**(name: String,groupMemberCount: Integer)--: Group pre **groupExists**: self.user.group -> select(n |n.Name = name) -> exists(name = true) pre **userMustExistInGroup**: self.user.numbers -> select(c | c.name = true or c.username = true) -> exists(name = true or username = true) post **deleteGroupMember**: let d = self.user.group -> size() = self.group @pre-> select(n |n.nbrOfPersons = groupMemberCount) -> size() - 1 in self.user.group @pre = self.group ->excluding(d) = true	joinGroup(name: String,groupMemberCount: Integer,g:Group)
removeFromGroup(a:Administrator, groupMemberCount: Integer) ***Taken from the Profile Class***	This allows a user to remove a participant from the group. userIsAnAdmin: A User has to be an admin in this case deleteGroupMember: A user is no longer a member of the group	**removeFromGroup**(a:Administrator, groupMemberCount: Integer) pre **userIsAnAdmin**: a.isAdmin = true post **deleteGroupMember**: let d = self.user.group -> size() = self.group @pre-> select(n |n.nbrOfPersons = groupMemberCount) ->	assignGroupAdmin() deleteGroupMembers()

		size() - 1 in self.group@pre = self.user.group ->excluding(d) = true	

makeCalls(nbrOfCallers: Integer, contactNumber: String, userName: String, name: String, c:Call) ***Taken from the Profile Class***	Allows the user to place a call. userMustBeOnline:A User must be online to make call accountMustExist: A user account must exist usernameMustBeUnique: A username must be unique contactNumberMustBeValid:A contact phone number must be valid	makeCalls(nbrOfCallers: Integer, contactNumber: String, userName: String, name: String, c:Call) pre **userMustBeOnline**: self.status = #Online pre **accountMustExist**: self.user.numbers -> forAll(c |c.username = userName or c.name= name) pre **usernameMustBeUnique**: self.user.contacts -> select(c |c.username = userName) -> isUnique(userName) pre **contactNumberMustBeValid**: self.user.numbers -> forAll(v | v.validNumber = true)	addContact(contactName: String, contactNumber: String)

	contactNumberMustBeUnique: A contact phone number must be unique	pre **contactNumberMustBeUnique**:self.user.numbers ->select(n|n.number = contactNumber) -> isUnique(contactNumber)	
	callTypeMustBeVoiceOrVideo: A phone call type must be either voice or video	pre **callTypeMustBeVoiceOrVideo**: self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video)	
	callIsAnsweredByContact: If call is answered by contact then a connection is established	post **callIsAnsweredByContact**: if self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video) then self.user.numbers ->forAll(n|n.number = contactNumber) and self.user.numbers.call -> forAll(s | s.status = #Connected) else self.user.numbers.call -> forAll(s | s.status = #Disconnected) endif	
	callHasEnded: A call has ended	post **callHasEnded**: if self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video) then self.user.numbers.call -> forAll(s | s.status = #Disconnected) else	

		self.user.numbers.call -> forAll(s | s.status = #HangupCall) endif	

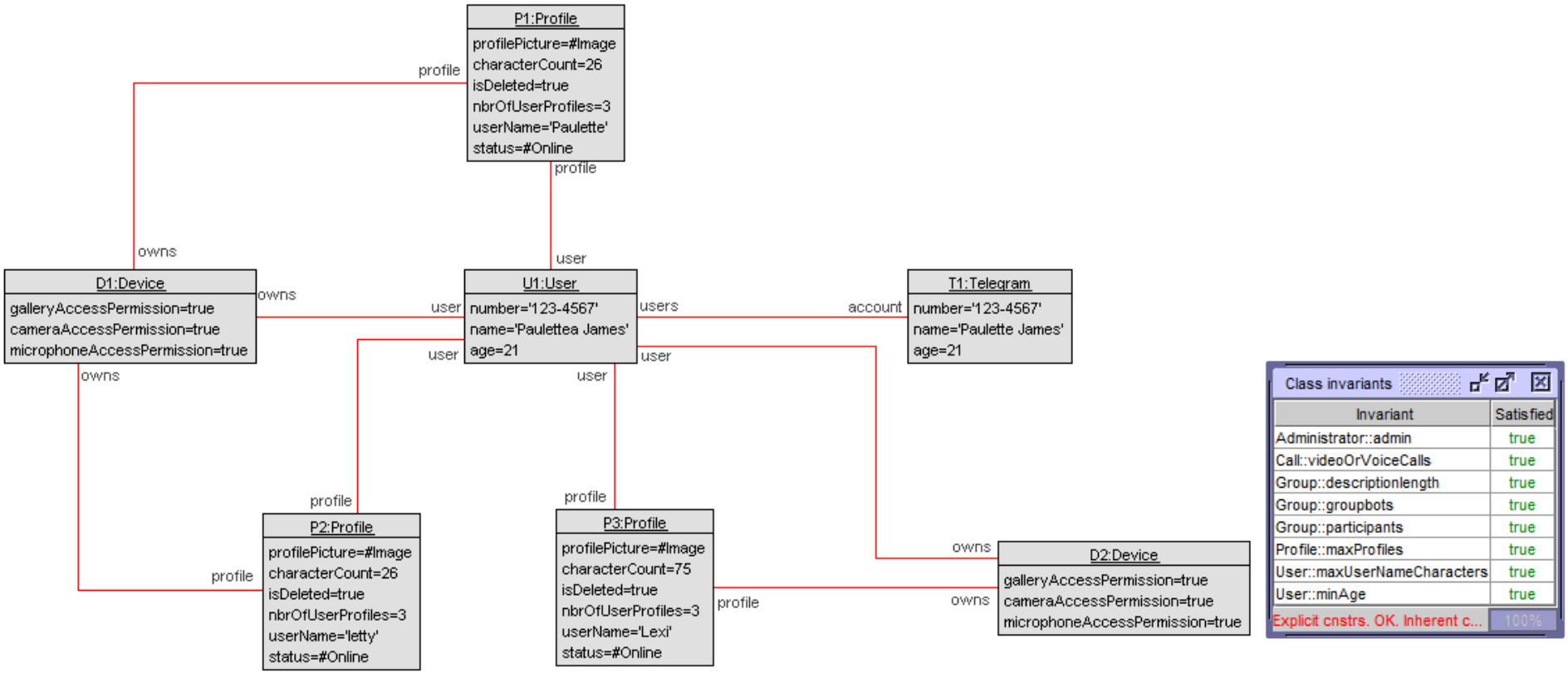
answerCall(c: Call)
Taken from the Profile Class
	This allows a user to answer and connect to a call.
	
	callTypeMustBeSetToRinging: A callType must be set to ringing
		answerCall(c: Call)
		
		pre **callTypeMustBeSetToRinging**:
		if c.callType = #Voice or c.callType = #Video
		then c.status = #Ringing
		else
		c.status = #Unanswered
		endif
		
		post **callConnectionIsEstablished**:
		if c.callType = #Voice or c.callType = #Video
			makeCalls(nbrOfCallers: Integer, contactNumber: String, userName: String, name: String, c:Call)

	callConnectionIsEstablished: A call connect is established	then c.status = #Connected
		else
		c.status = #Disconnected
		endif	

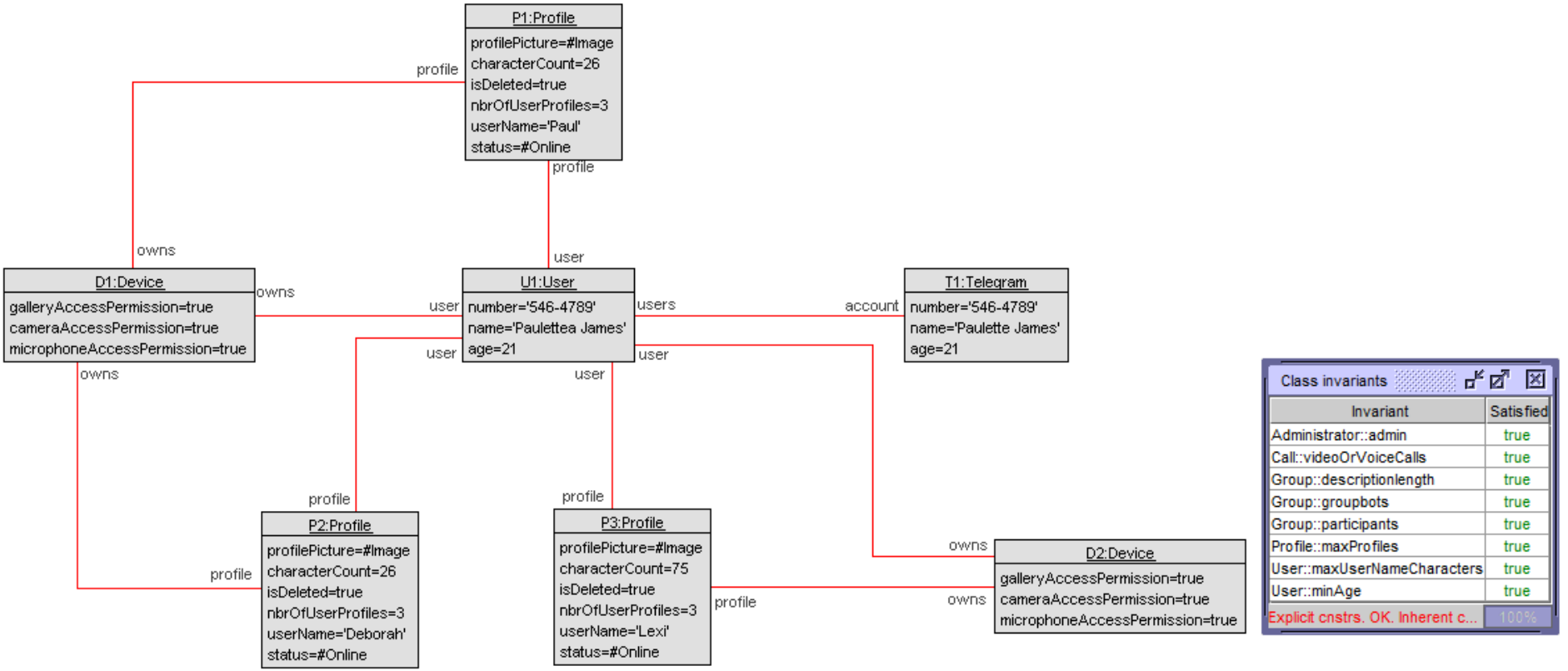
endCall(c: Call)
Taken from the Profile Class
	This allows for a user to end and disconnect a call.
	
	callConnectionIsEstablished : A call connection is established
	
	callHasEnded: A call is ended
		endCall(c: Call)
		
		post **callConnectionIsEstablished**:
		if c.callType = #Voice or c.callType = #Video
		then c.status = #Connected
		else
		c.status = #Disconnected
		endif
		
		post **callHasEnded**:
		if c.callType = #Voice or c.callType = #Video
		then c.status = #HangupCall
		else
		c.status = #Disconnected
		endif
		end
			answerCall(c: Call)

Simulation of Use Case

Simulation of the Update User Profile Object Model

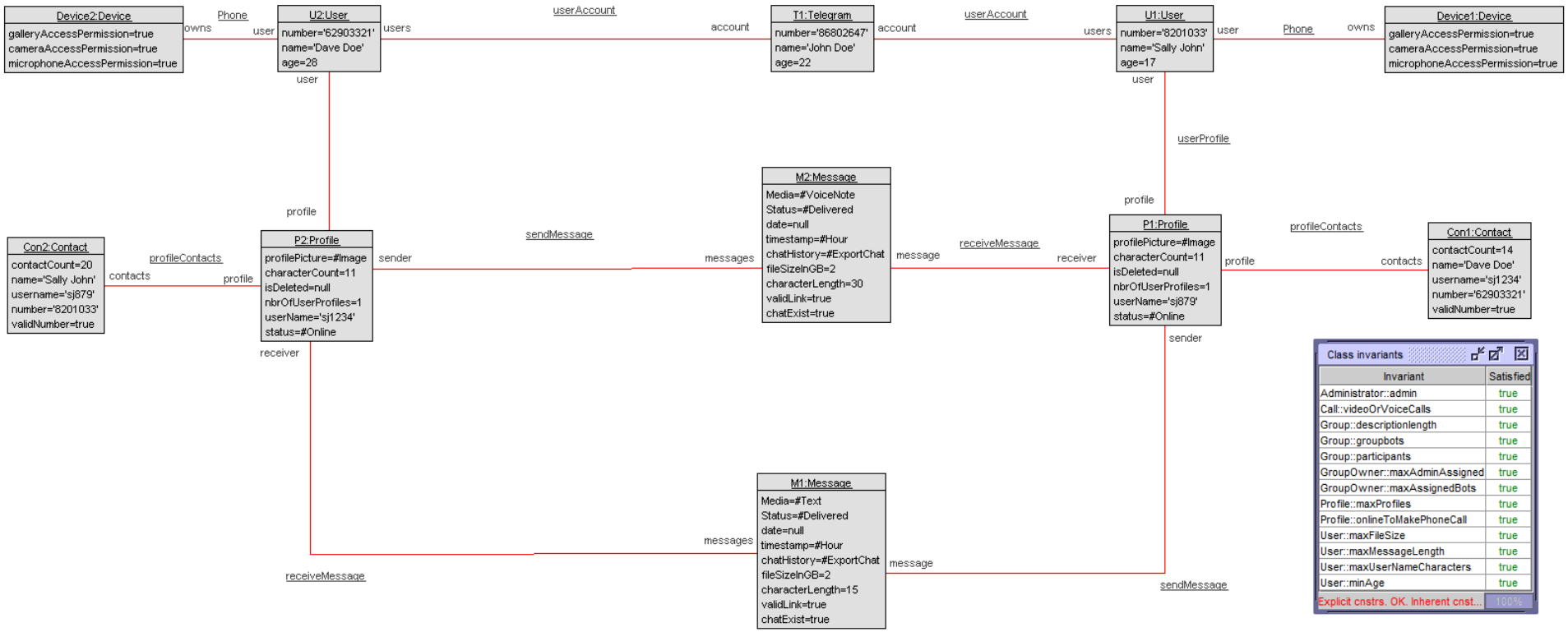


The object diagram above represents a user of Telegram (Pauletta James). She has three active profiles within Telegram, all consisting of different usernames. The diagram illustrates how a user of Telegram can have up to three user profiles all having unique usernames. It is also seen where a user can have multiple devices and access any of their profiles from any device.



The diagram above represents the updates the Telegram user (Pauletta James) from the previous diagram made to her user profiles, it is seen where Pauletta changed the phone number associated with her account and changed her profile usernames for profile 1 and 2.

Simulation of the Send Message Object Model



The object diagram above illustrates two users corresponding via text message. User1 (Sally John) is corresponding with User2 (Dave Doe) they are both corresponding user their Profile number 1.

User 1 and 2 are connected to their own mobile devices with permissions allowed. User 1 then sends a message, M1 To User 2. M1 is then received by User2's profile. The second users profile replies and now sends a message (M2) to the first users profile. M2 is received by the profile 1 (P1) which belongs to the User 1.

Make Call Use Case

9.1

Operation 1: Taken From Profile Class

```
makeCall(contactNumber: String, userName: String, name: String): Contact
        begin
                declare c: Contact;
                c := new Contact();
                c.name := name;
                c.username:= userName;
                c.number := contactNumber;
                c.viewContact(contactNumber,userName,name);
                --insert(self,c) into profileContacts;
                result := c;
        end
```

Operation 2: Taken From Profile Class

```
searchContact(contactNumber: String, userName: String, name: String): Contact
        begin
                declare c: Contact;
                c := new Contact();
                c.name := name;
                c.username:= userName;
                c.number := contactNumber;
                --c.viewContact(contactNumber,userName,name);
                self.makeCall(contactNumber,userName,name);
                --insert(self,c) into profileContacts;
                result := c;
        end
```

Operation 3: Taken From Telegram class

```
createAccount(name: String,number: String, age: Integer): User
        begin
                declare u: User;
                u := new User();
                u.name := name;
                u.number:= number;
                u.age:= age;
                u.addProfile(name,number,age);
                --insert(self,u) into userAccount;
                result := u;
        end
```

Operation 4: Taken From User Class

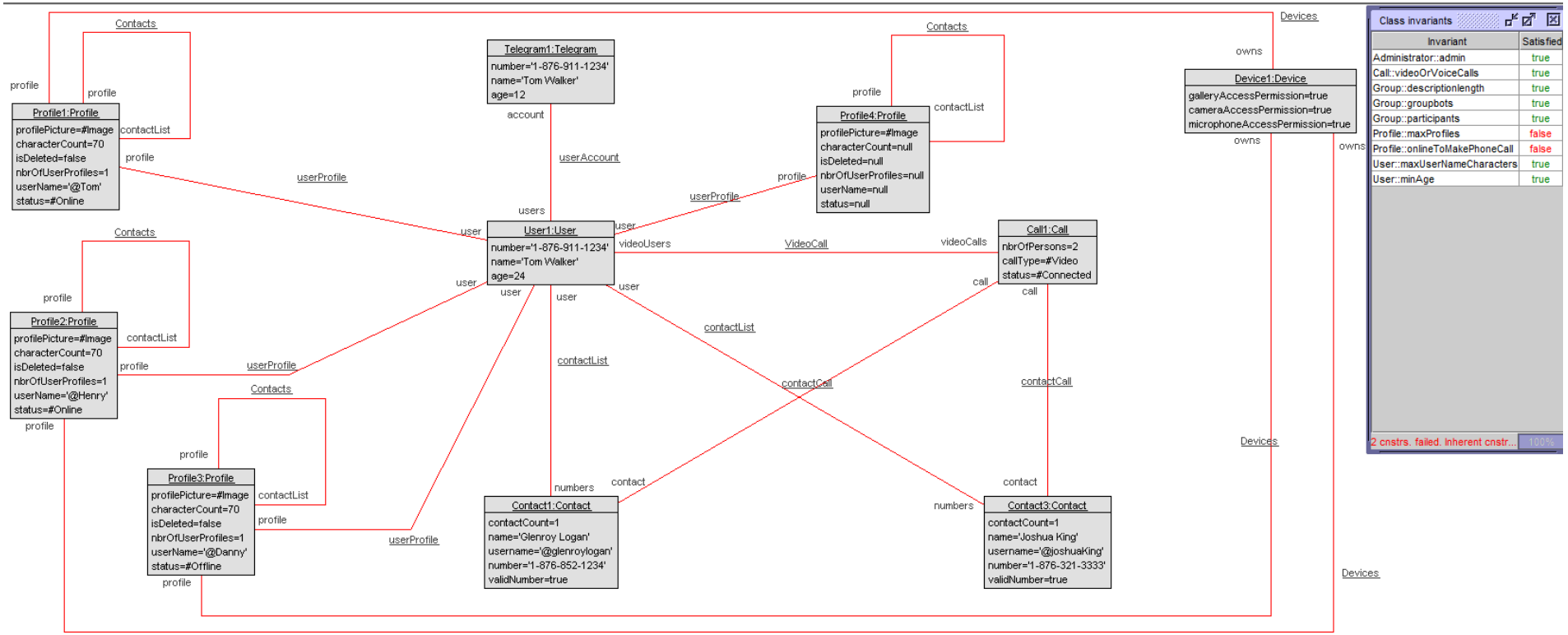
```
addProfile(name: String, number: String, age: Integer)--: Profile
        begin
                self.name := name;
                self.number := number;
                self.age := age;
        end
```

Operation 5: Taken From Contact Class

```
viewContact(contactNumber: String,userName: String ,name: String)
        begin
                self.name:= name;
                self.number:= contactNumber;
                self.username:= userName;
        end
```

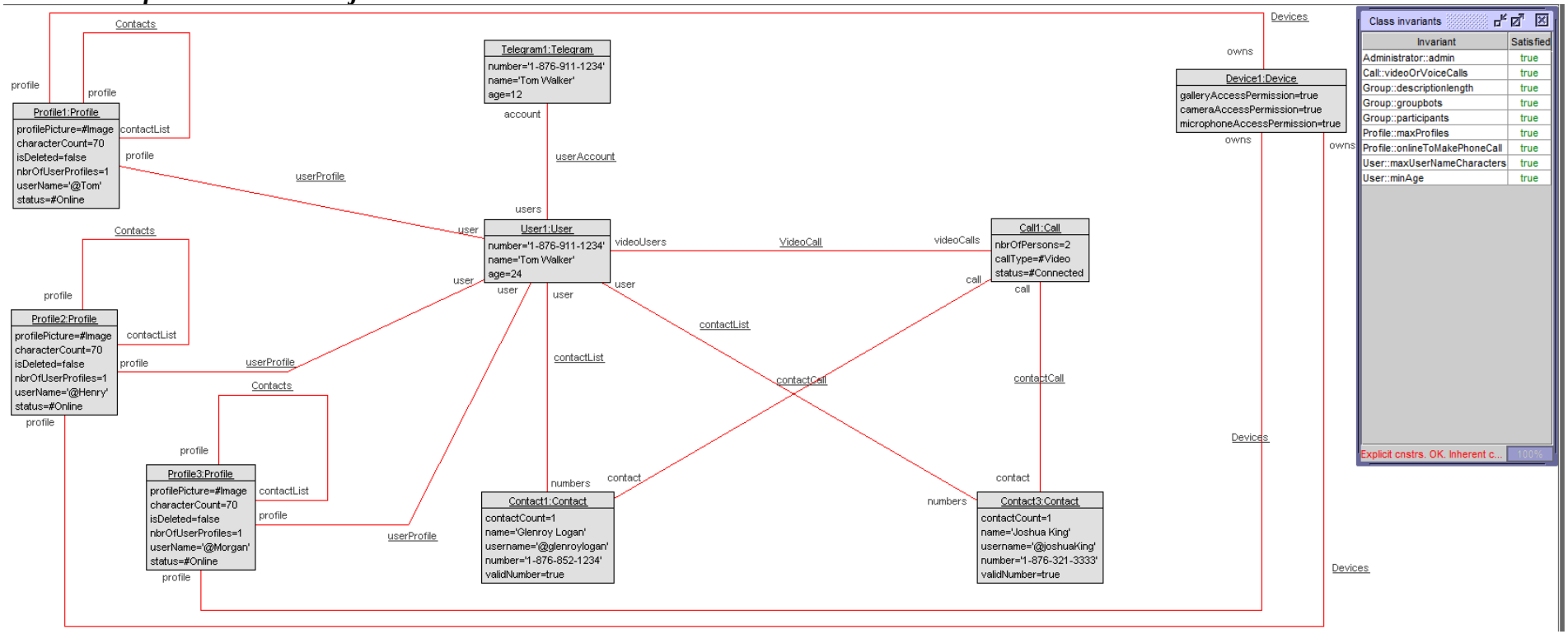
9.2
For the test criteria for predicate coverage where the predicate evaluates to false

Violated Model is in the screenshot below:
Violation 1: A user cannot have more than 3 profiles the diagram below shows a user having 4 profiles
Violation 2: A user registering for telegram cannot be below 16 years of age



9.3
For the test criteria for predicate coverage where the predicate evaluates to true

9.3.1 Show the test case, i.e., the object model
Simulation of the Make Call Object Model



The Make Call Operation is the operation that is being tested:

makeCall(contactNumber: String, userName: String, name: String): Contact

```
begin
    declare c: Contact;
    c := new Contact();
    c.name := name;
    c.username:= userName;
    c.number := contactNumber;
    c.viewContact(contactNumber,userName,name);
    --insert(self,c) into profileContacts;
    result := c;
end
```

Make Call Pre and Post Conditions are below:

--User must be online to make call

pre userMustBeOnline: self.status = #Online

--account must exist

pre accountMustExist: self.user.numbers -> forAll(c |c.username = userName or c.name= name)

--username must be unique

pre ***usernameMustBeUnique***: self.user.numbers -> select(c |c.username = userName) -> isUnique(userName)

--contact phone number must be valid

pre ***contactNumberMustBeValid:*** self.user.numbers -> forAll(v | v.validNumber = true)

--contact phone number must be unique

pre ***contactNumberMustBeUnique:*** self.user.numbers ->select(n|n.number = contactNumber) -> isUnique(contactNumber)

--phone call type must be either voice or video

pre ***callTypeMustBeVoiceOrVideo:*** self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video)

--if call is answered by contact then a connection is established

post callIsAnsweredByContact:

if self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video)

then self.user.numbers ->forAll(n|n.number = contactNumber) and self.user.numbers.call -> forAll(s | s.status = #Connected)

else

self.user.numbers.call -> forAll(s | s.status = #Disconnected)

endif

--call has ended

post callHasEnded:

if self.user.audioCalls -> forAll(c | c.callType = #Voice) or self.user.videoCalls -> forAll(c | c.callType = #Video)

then self.user.numbers.call -> forAll(s | s.status = #Disconnected)

else

self.user.numbers.call -> forAll(s | s.status = #HangupCall)

endif

9.3.2 explain the expected result

For the operation **makeCalls**:

The precondition is a conjunction of all the precondition parts, i.e., userMustBeOnline and accountMustExist and usernameMustBeUnique, contactNumberMustBeValid, contactNumberMustBeUnique and callTypeMustBeVoiceOrVideo =

status =#Online and
c.username = userName = true or c.name = name = true and
c.username = userName and
v.validNumber = true and
n.number = contactNumber and
c.callType =#Voice or c.callType = #Video

The postcondition is a conjunction of all the postcondition parts, i.e., callIsAnsweredByContact and callHasEnded =

c.callType =#Voice or c.callType = #Video then c.status =#Disconnected else c.status =#Hangup and
c.callType =#Voice or c.callType = #Video then c.status =#Ringing else c.status =#Unanswered

Lets' call the precondition preB and the postcondition postB

preB has five (8) clauses. The clauses are:

A. status = online
B. username = userName
C. name = name
D. username = userName
E. validNumber = true
F. number = contactNumber
G. callType = Voice
H. callType = Video

final expression:
a & (b |c) & d & e & f & (g | h)

Test Criteria

1. preB is true
2. preB is false
3. clause 1 is true
4. clause 1 is false
5. clause 2 is true
6. clause 2 is false
7. clause 3 is true
8. clause 3 is false
9. clause 4 is true
10. clause 4 is false
11. clause 5 is true
12. clause 5 is false
13. clause 6 is true
14. clause 6 is false

Satisfying Predicate Coverage

For p = true, row 1 I

For p = false, can choose any row from 3,5,7,9...127

Truth Table:

Row#	a	b	c	d	e	f	g	h	P
1	T	T	T	T	T	T	T	T	T
2	T	T	T	T	T	T	T		T
3	T	T	T	T	T	T		T	T
4	T	T	T	T	T	T			
5	T	T	T	T	T		T	T	
6	T	T	T	T	T		T		
7	T	T	T	T	T			T	
8	T	T	T	T	T				
9	T	T	T	T		T	T	T	
10	T	T	T	T		T	T		
11	T	T	T	T		T		T	
12	T	T	T	T		T			
13	T	T	T	T			T	T	
14	T	T	T	T			T		
15	T	T	T	T				T	
16	T	T	T	T					
17	T	T	T		T	T	T	T	
18	T	T	T		T	T	T		
19	T	T	T		T	T		T	
20	T	T	T		T	T			
21	T	T	T		T		T	T	
22	T	T	T		T		T		
23	T	T	T		T			T	
24	T	T	T		T				
25	T	T	T			T	T	T	
26	T	T	T			T	T		
27	T	T	T			T		T	
28	T	T	T			T			
29	T	T	T				T	T	
30	T	T	T				T		
31	T	T	T					T	
32	T	T	T						
33	T	T		T	T	T	T	T	T
34	T	T		T	T	T	T		T

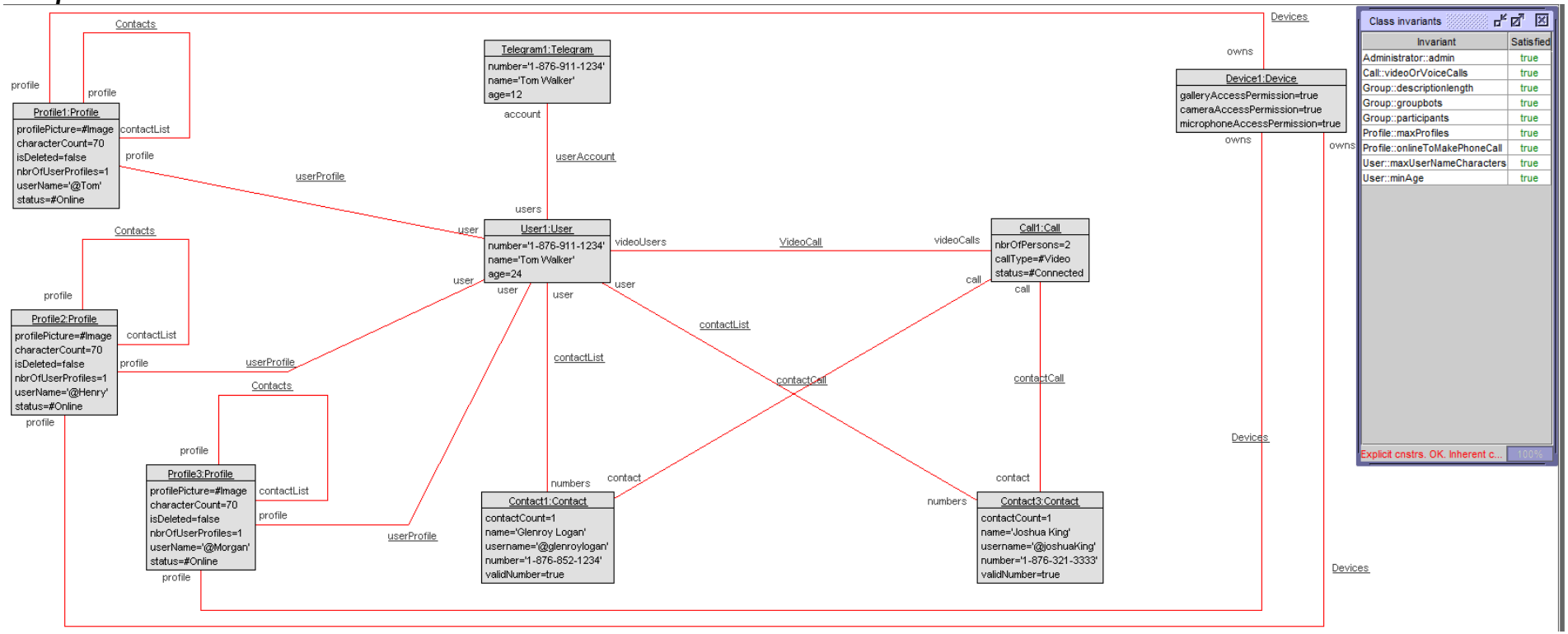
Satisfying Clause Coverage

row 1, for a = true, b = true, c = true, d = true, e = true, f = true, g = true, h= true
row 128 for a = false, b = false, c = false, d = false, e = false, f = false, g = false, h= false

107	T			T		T		T	
108	T			T		T			
109	T			T			T	T	
110	T			T			T		
111	T			T				T	
112	T			T					
113	T				T	T	T	T	
114	T				T	T	T		
115	T				T	T		T	
116	T				T	T			
117	T				T		T	T	
118	T				T		T		
119	T				T			T	
120	T				T				
121	T					T	T	T	
122	T					T	T		
123	T					T		T	
124	T					T			
125	T						T	T	
126	T						T		
127	T							T	
128	T								
129		T	T	T	T	T	T	T	
130		T	T	T	T	T	T		
131		T	T	T	T	T		T	
132		T	T	T	T	T			
133		T	T	T	T		T	T	
134		T	T	T	T		T		
135		T	T	T	T			T	
136		T	T	T	T				
137		T	T	T		T	T	T	
138		T	T	T		T	T		
139		T	T	T		T		T	
140		T	T	T		T			
141		T	T	T			T	T	
142		T	T	T			T		
143		T	T	T				T	

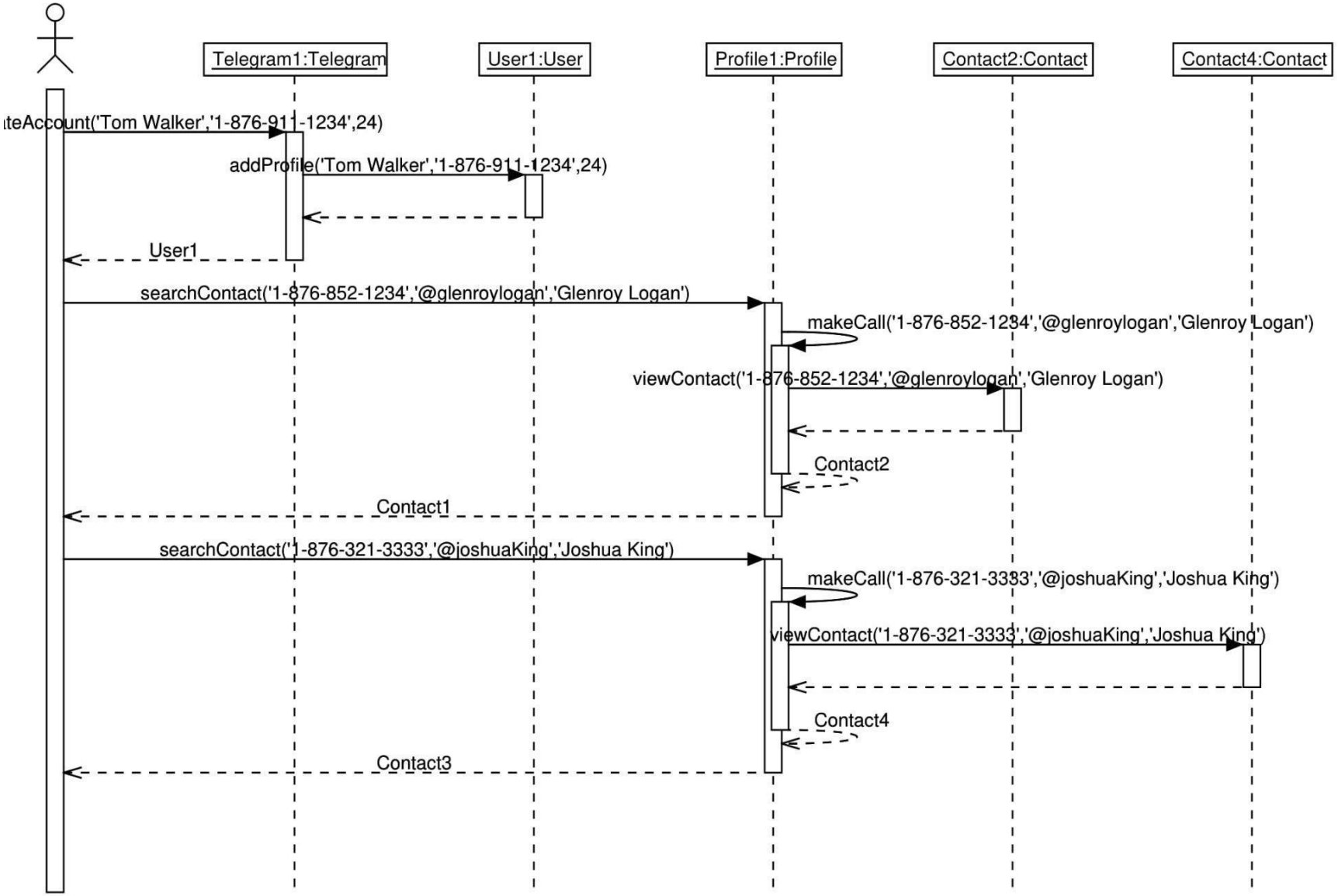
9.3.3

The resulting object model when the simulation completed Satisfied Model is in the screenshot below:



9.4

Sequence Diagram Of The Object Model



Protocol state machines (PSM)

Update Profile Settings PSM

Figure of the class state machine is created for

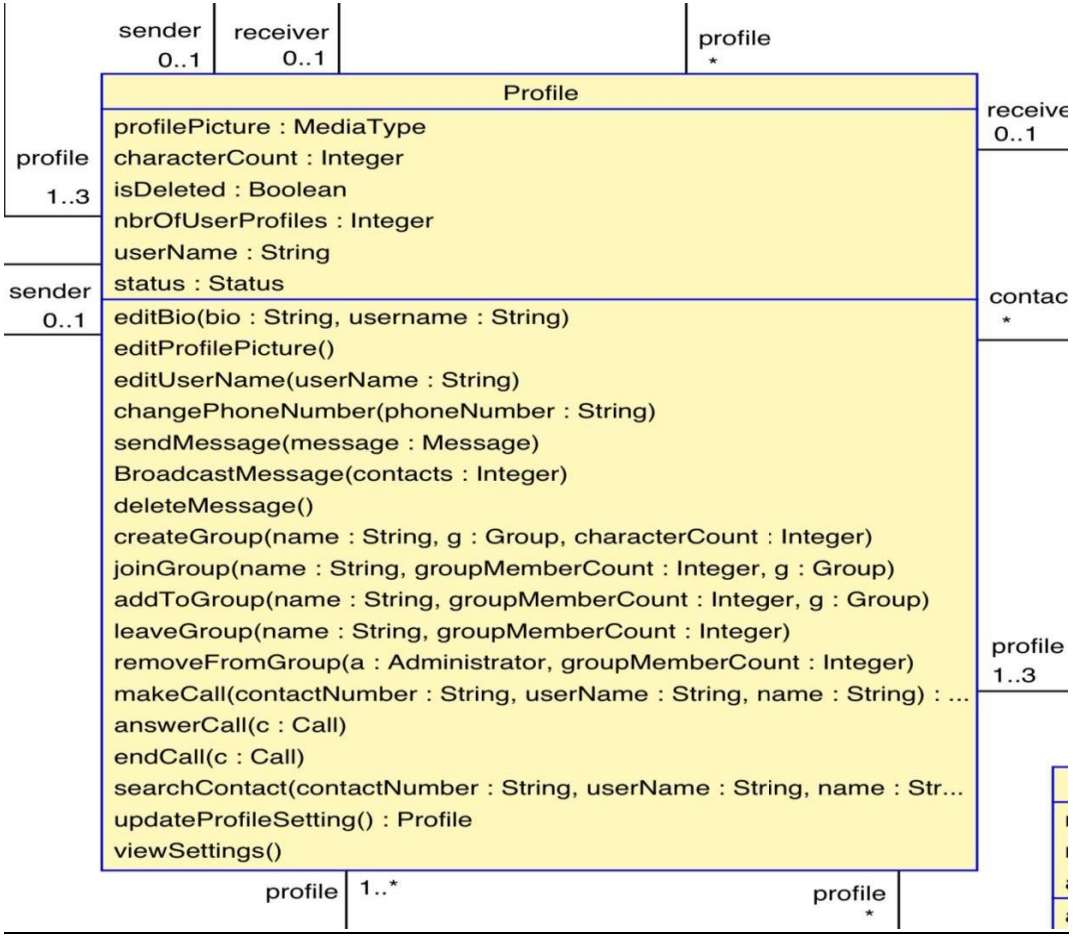
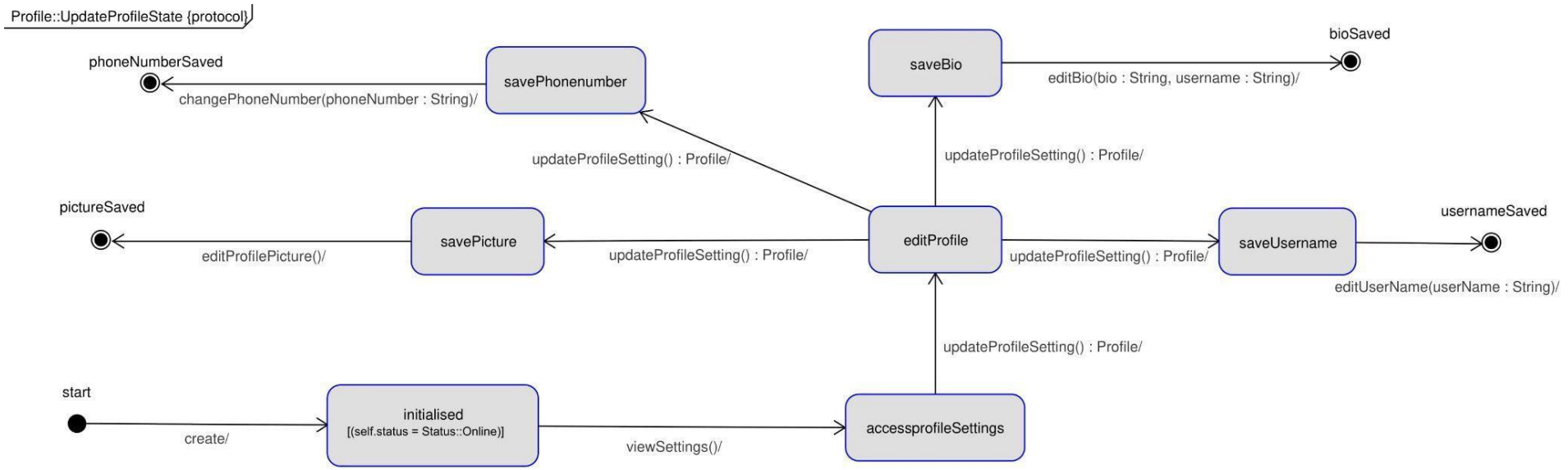


Figure of Update Profile Settings PSM



USE code to create it

```
statemachines
	psm UpdateProfileState
	states
		start: initial

		initialised [status = #Online]
		accessprofileSettings
		editProfile
		saveBio
		savePicture
		saveUsername
		savePhonenumber
		bioSaved: final
		pictureSaved: final
		usernameSaved: final
		phoneNumberSaved: final
	transitions
		start -> initialised {create}
		//initialised -> initialised{}
		initialised -> accessprofileSettings{viewSettings()}

		accessprofileSettings -> editProfile{updateProfileSetting()}
		editProfile -> saveBio{updateProfileSetting()}
		editProfile -> savePicture{updateProfileSetting()}
		editProfile -> saveUsername{updateProfileSetting()}
		editProfile -> savePhonenumber{updateProfileSetting()}

		saveBio -> bioSaved{editBio()}
		savePicture -> pictureSaved{editProfilePicture()}
		saveUsername -> usernameSaved{editUserName()}
		savePhonenumber -> phoneNumberSaved{changePhoneNumber()}
	end
```

English description of the protocol state machine

This protocol state machine is within the profile class of this system. It demonstrates the different states and transitions involved in a user updating different profile settings. The initial state of this psm is when the user is online, therefore the User status was set to online. From there we can transition and go to the edit profile settings state which allows a user to view their settings. Here there are different states that follow, firstly, we have the edit Profile State. At this state we can update your Profile picture and then save it or we can update the username and then save that one as well. After doing either one of those tasks we reach the final state which is when the new profile picture is saved or the new username was saved. Additionally, from this edit Profile state it is possible to edit your number attached to your profile. From the change profile number state we have a final state which indicates that the new phone number is saved.Lastly, we can navigate to our next state which allows you to edit your profile bio and upon doing so we transition to the final state where the new edited bio is saved to your Telegram profile.

Figure of class state machine is created for

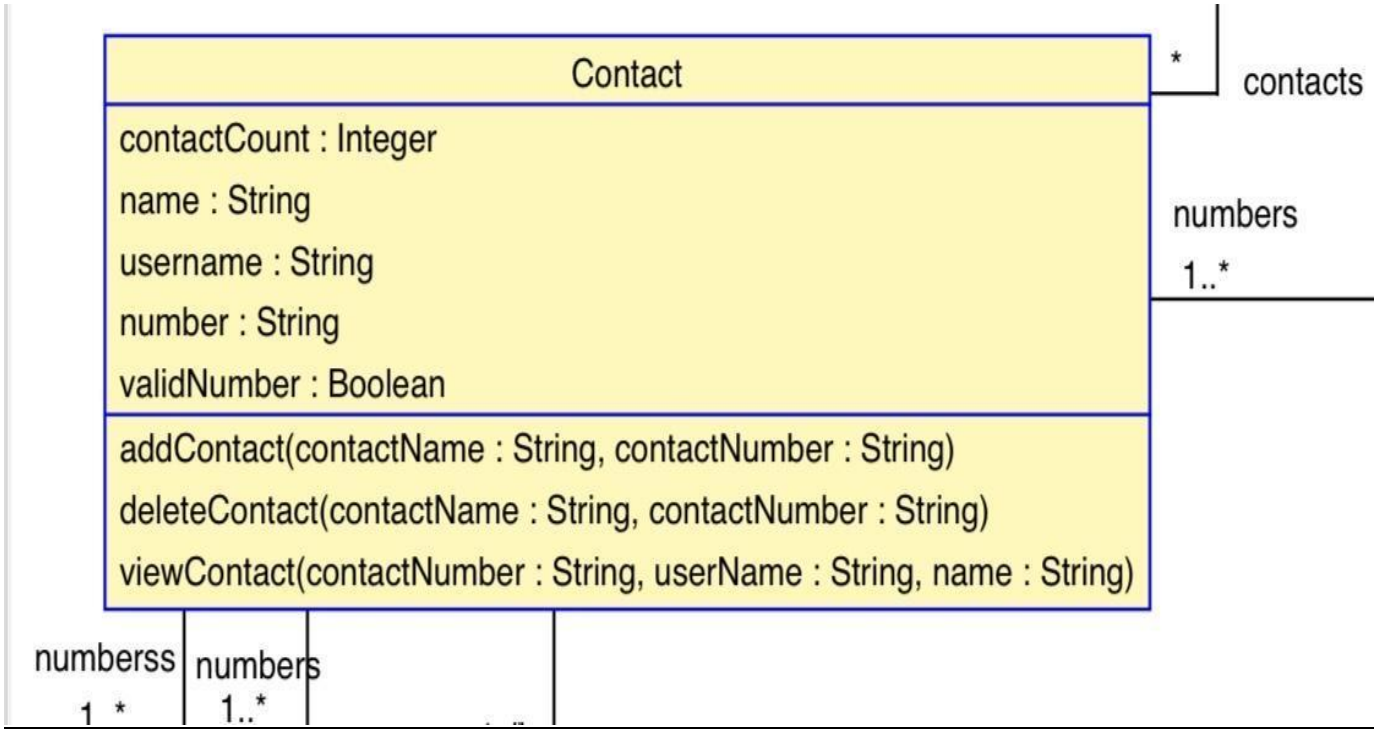
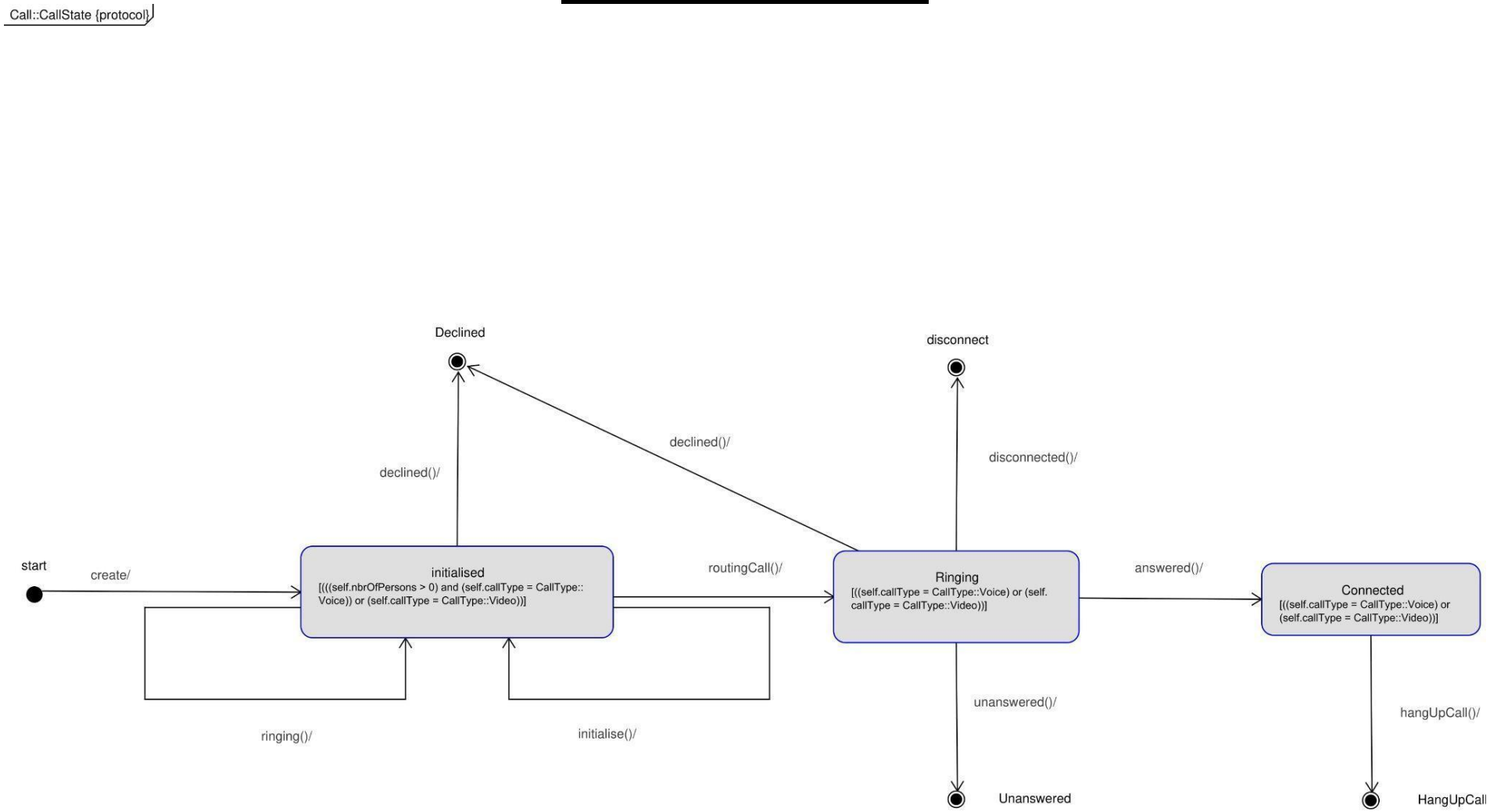


Figure of Make Call PSM



USE code to create it

```
statemachines
psm CallState
states
   start: initial
                initialised [nbrOfPersons > 0 and callType = #Voice or callType = #Video]
                Ringing [callType = #Voice or callType = #Video]
                Connected [callType = #Voice or callType = #Video]
                disconnect: final
                Declined: final
                HangUpCall: final
                Unanswered: final
transitions
   start -> initialised {create}

                initialised -> initialised {ringing()}
                initialised -> initialised {initialise()}
                initialised -> Declined {declined()}
                initialised -> Ringing {routingCall()}

                Ringing -> Connected{answered()}
                Ringing -> Declined {declined()}
                Ringing -> Unanswered {unanswered()}
                Ringing -> disconnect {disconnected()}

                Connected -> HangUpCall {hangUpCall()}
end
```

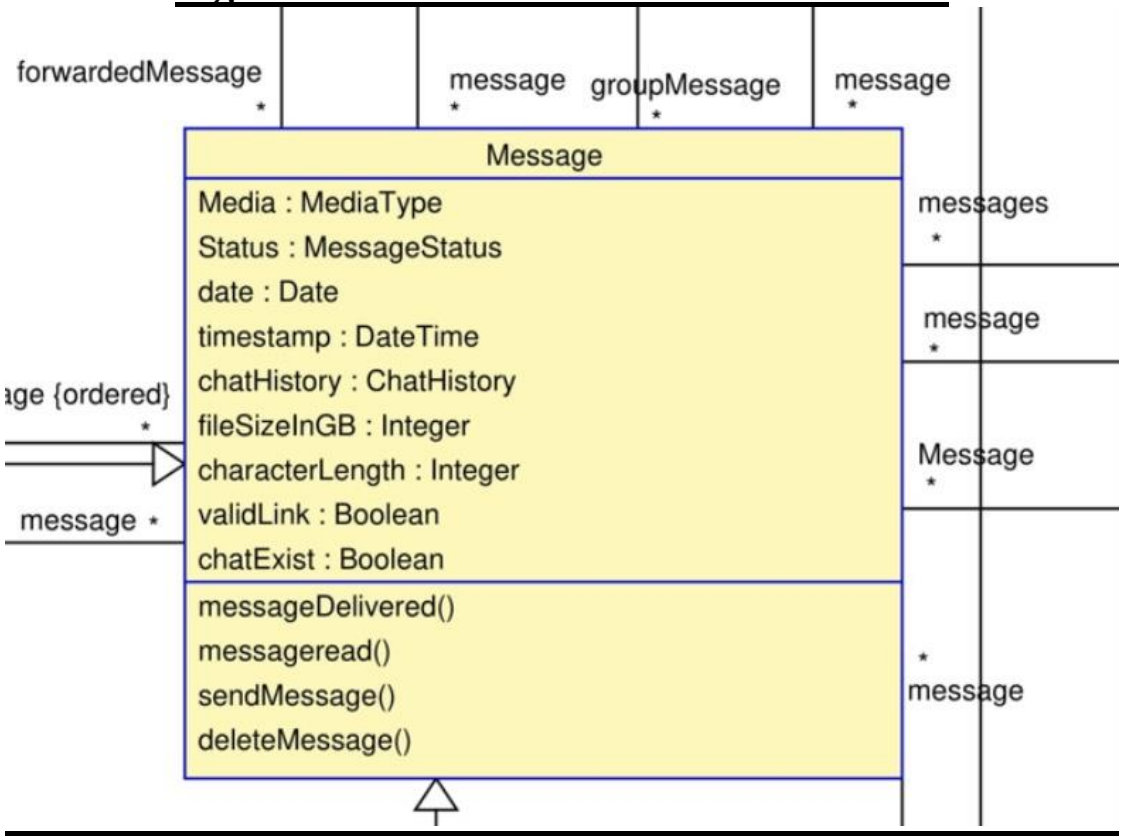
English description of the protocol state machine

This state machine is within the call class of the Telegram system model. We begin navigating from the start state to the initialized state and at this state there must be more than 0 persons to make the call and the call type is either a video or audio call. At this state we have the ringing

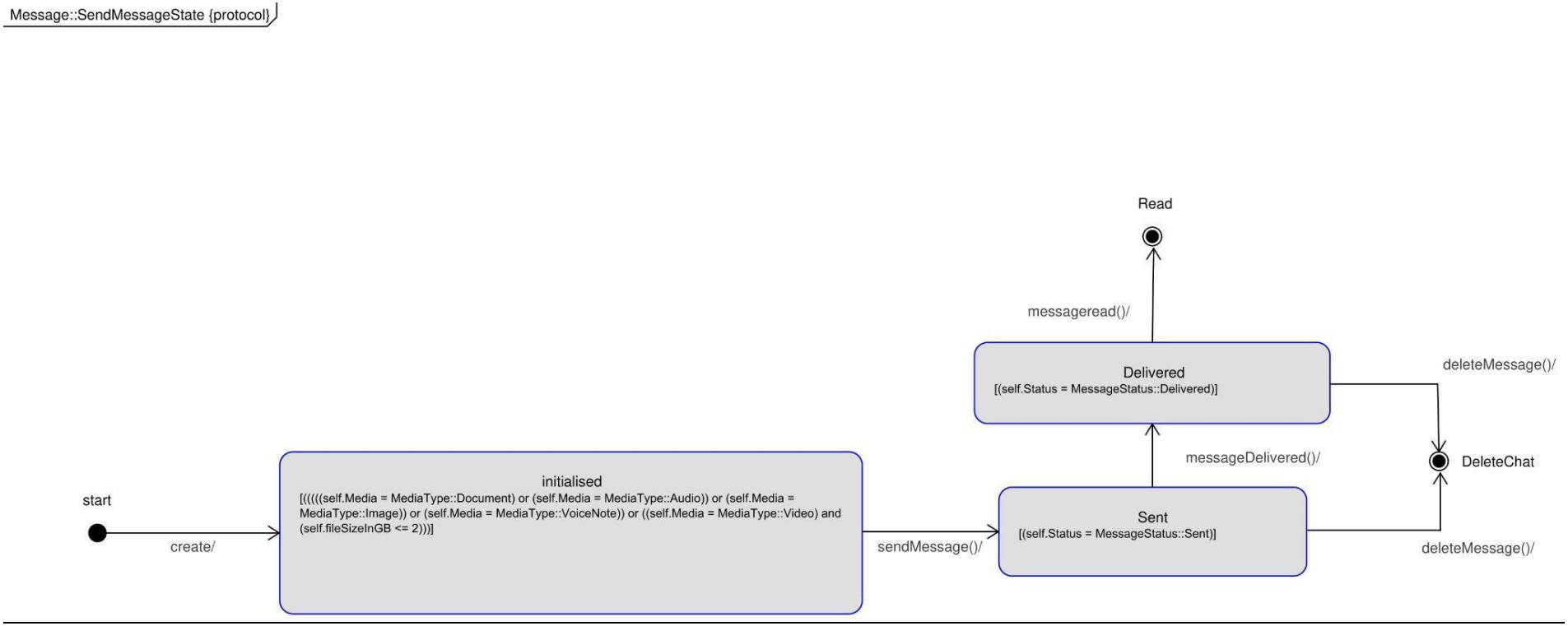
From here we navigate to the Ringing state by the routingCall(). At this point the call can take three final states; it can either be declined, disconnected or unanswered. At this state the same condition is applied where the call type is either a video or audio call. If we do not reach one of the three mentioned final states we then move to the Connected state. Here, the condition is applied again and the call type is either a video or audio call. At this state there is only one final state we can go to which is the hang up call state which is done when the call is completed.

Send Message PSM

Figure of class state machine is created for



Send Message PSM



USE code to create it

```
statemachines
psm SendMessageState
        states
      start: initial
      initialised [self.Media = MediaType::Document or self.Media = MediaType::Audio or self.Media = MediaType::Image or self.Media =
MediaType::VoiceNote or self.Media = MediaType::Video and self.fileSizeInGB <= 2]
                  Sent [Status = #Sent]
                  Delivered [Status = #Delivered]
      DeleteChat:final
      Read:final
   transitions
      start -> initialised {create}
      initialised -> Sent {sendMessage()}
      Sent ->  Delivered {messageDelivered()}
      Sent -> DeleteChat {deleteMessage()}
      Delivered -> DeleteChat {deleteMessage()}
      Delivered -> Read {messageread()}
   end
```

English description of the Send Message protocol state machine

This is the protocol state machine for Send Message and it is located in the Message class. We begin at the start and go to the initialized state where the condition here is that the message has to be one of the approved Telegram message types. The approved media types are documents, audio, video, images, voicenotes. The message would also have to be under 2GB in size since that is the maximum size of messages allowed in Telegram. From the initialized state we can navigate to the Sent state by sendMessage(). Here, the condition requires that the message status will be set to sent and from here we can naivagte to a final state called Delete message. From the Sent state we can also navigate to the delivered state where the condition is that message status has to be set to be delivered.At this point the receiver of the message can Read it or the sender can delete it. This is shown by the final states Read and Delete respectively.

Post mortem

Throughout the course of this semester-long project, we got to investigate a familiar application we all use. Especially now, given the current circumstances, we rely on tools like these to communicate and keep informed. This project challenged us in more ways than one whether it was in the constant reevaluating of our original ideas, researching, manipulating this new (OCL) environment, seeking advice from our peers or lecturer, preparing and rehearsing for presentations or even internet connectivity issues, all while balancing other assignments; we greatly benefitted from our experience in this course and garnered teamwork skills that will ultimately make us better software engineers and adequately prepare us for the real world. All the concepts learnt from this course will be vital in helping us as software engineers to be able to apply our skills to real world situations and thus help to further develop ourselves as well as society in general.