# 3. PASSPORT AUTOMATION SYSTEM

**Aim:** To write problem statement, discover requirements and create UML models for automation of Passport issue process toward designing Passport Automation System.

**Problem Statement:**

Passport Automation System is used in the effective dispatch of passport to all of the applicants. This system adopts a comprehensive approach to minimize the manual work (application, verification, issue, re-issue, dispatch and seize services) and schedule resources, time in a cogent manner. The core of the system is to get the online registration form (with details such as name, address etc.,) filled by the applicant whose testament is verified for its genuineness by the Passport Automation System with respect to the already existing information in the database.

**Requirements Specification: ( http://vlabs.iitkgp.ernet.in/se/1/theory/ )**
Requirements identification is the first step of any software development project. Requirements specify how the target system should behave. In this experiment we will learn how to identify functional and non-functional requirements from a given problem statement. Requirements can be classified into two groups based on what they describe:

- **Functional requirements (FRs):** These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.
- **Non-functional requirements (NFRs):** These could typically define how the system should behave under certain situations.

Once all the functional and non-functional requirements have been identified, they are documented formally in software requirements specification (SRS), which then serves as a legal agreement between client and software company.

**Identification of functional requirements**
Form the above problem statement, even without doing any deep analysis, we might easily

identify some of the basic functionality of the system:

New user registration, user login, apply, check status, scrutiny, initiate verification, grant

appointment, verify, issue, re-issue, dispatch, seize.

**Identification of non-functional requirements**
Let's try to identify a few non-functional requirements.

- **Performance Requirements:**
    - This system should remain accessible 24x7

    - At least 50 users should be able to access the system altogether at any given time
- **Security Requirements:**
    - The database of passport automation (PA) should not store any password in plain text -- a hashed value has to be stored
- **Software Quality Attributes**
- **Database Requirements**
- **Design Constraints:**
    - The PA has to be developed as a web application, which should work with Firefox 5, Internet Explorer 8, Google Chrome 12, Opera 10
    - The system should be developed using HTML 5

**UML models creation:**
Using Star UML, a new model is opened and renamed as PassportAutomation. The following diagrams are attempted to represent the requirements specified above.

## 1. Use case Diagram **( http://vlabs.iitkgp.ernet.in/se/3/theory/ )**

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

**Actor**
An actor can be defined as an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems.

**Use Case**

A use case is simply a functionality provided by a system. In passport automation system, issue is a functionality that the system provides. Therefore, this is a use case. Other possible use cases includes, apply, verify, re-issue, and so on.

**Subject**

Subject is simply the system under consideration. Use cases apply to a subject. For example, an Passport Automation is a subject, having multiple use cases, and multiple actors interact with it.

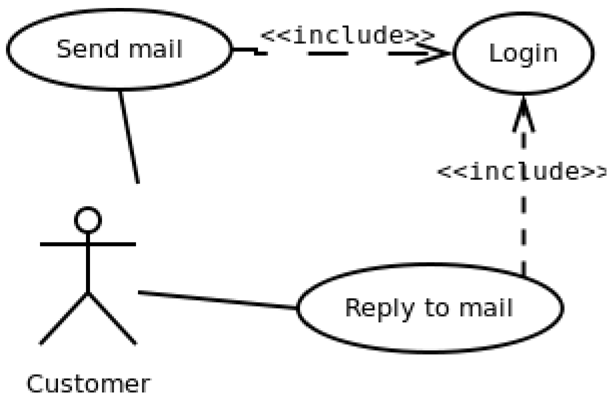
**Use Case Relationships**

Three types of relationships exist among use cases:

- Include relationship
- Extend relationship
- Use case generalization

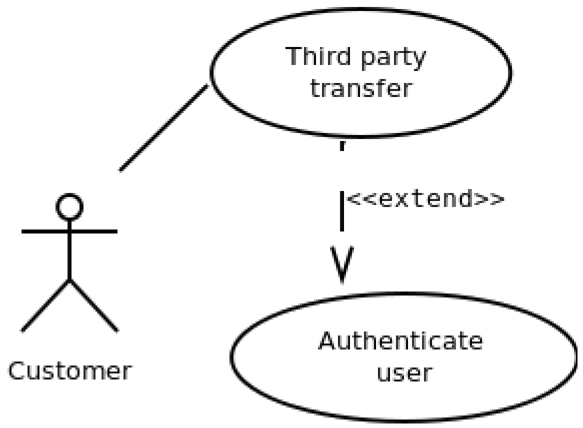
**Include Relationship**

Include relationships are used to depict common behaviour that are shared by multiple use cases. For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a *login* use case, which is included by *compose mail*, *reply*, and *forward email* use cases. Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.



**Extend Relationship**

Use case extensions are used used to depict any variation to an existing use case. They are used to the specify the changes required when any assumption made by the existing use case becomes false. Let's consider an online banking. The system allows an authenticated user to do. Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.



**Identifying Actors**

Given a problem statement, the actors could be identified by asking the following questions :

- Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor)
- Who keeps the system working? (This will help to identify a list of potential users)
- What other software / hardware does the system interact with?
- Any interface (interaction) between the concerned system and any other system?

**Identifying Use cases**

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "check status".

## 2. Class diagram

It is a graphical representation for describing a system in context of its static construction.

Class diagram contains the system classes with its data members, operations and relationships between classes.

Class

A set of objects containing similar data members and member functions is described by a class. In UML syntax, class is identified by rectangle with three compartments which contain

- **Class name**

  A class is uniquely identified in a system by its name. A textual string is taken as class name. It lies in the first compartment in class rectangle.

- **Attributes**

  Property shared by all instances of a class. It lies in the second compartment in class rectangle.

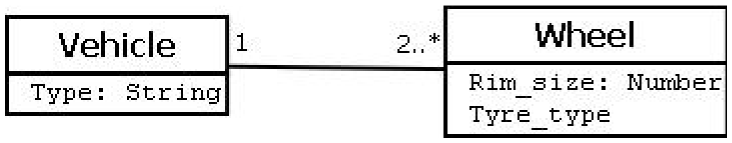
- **Operations**

  An execution of an action can be performed for any object of a class. It lies in the last compartment in class rectangle.

**Multiplicity:**

It describes ow many numbers of instances of one class is related to the number of instances of another class in an association.

**Notation for different types of multiplicity:**



## 3. Sequence diagram

It represents the behavioral aspects of a system. Sequence diagram shows the interactions between the objects by means of passing messages from one object to another with respect to time in a system.

Sequence diagram contains the objects of a system and their life-line bar and the messages passing between them.

Object

Objects appear at the top portion of sequence diagram. Object is shown in a rectangle box. Name of object precedes a colon ':' and the class name, from which the object is instantiated. The whole string is underlined and appears in a rectangle box. Also, we may use only class name or only instance name.

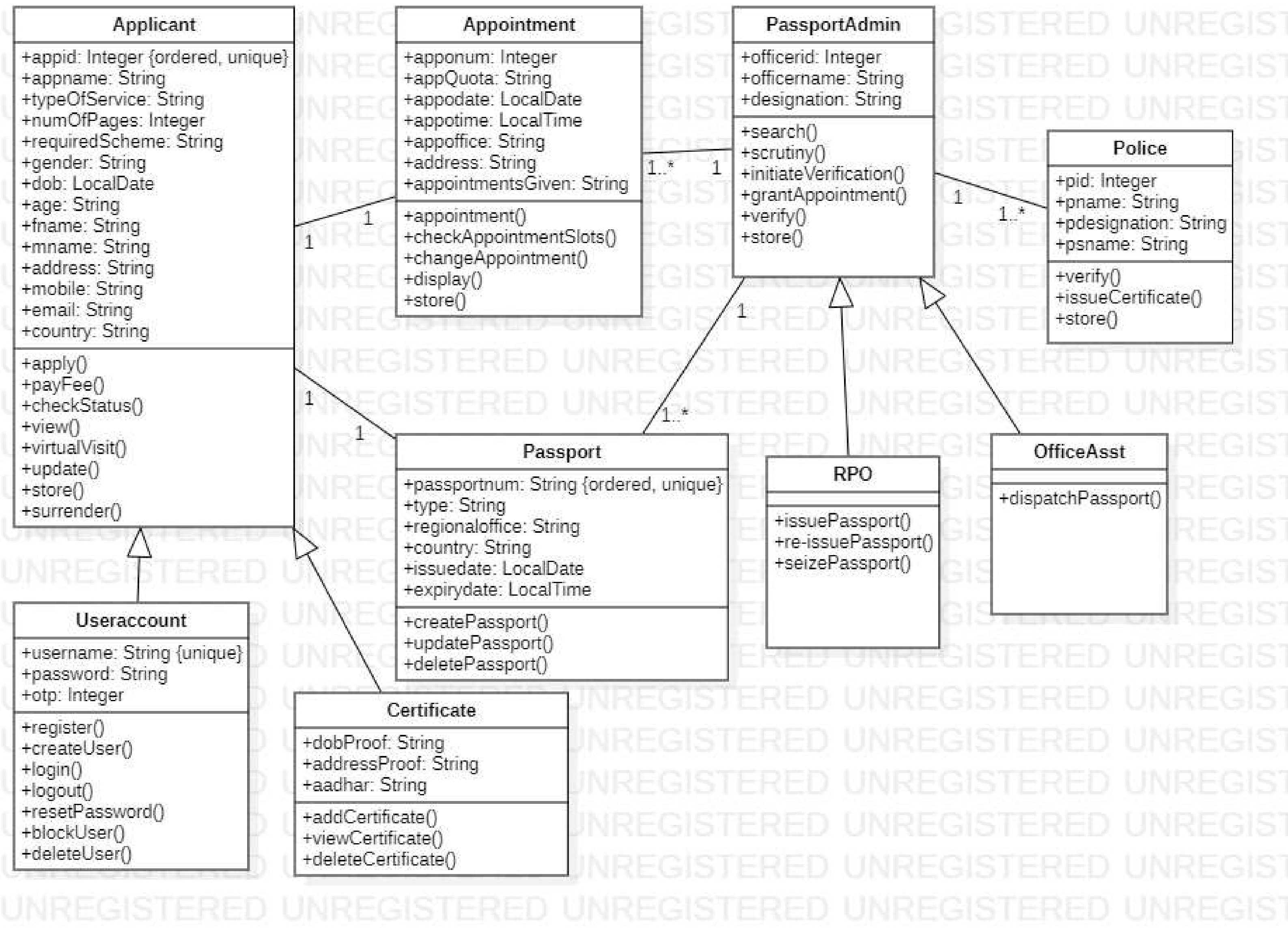
Objects which are created at the time of execution of use case and are involved in message passing , are appear in diagram, at the point of their creation.

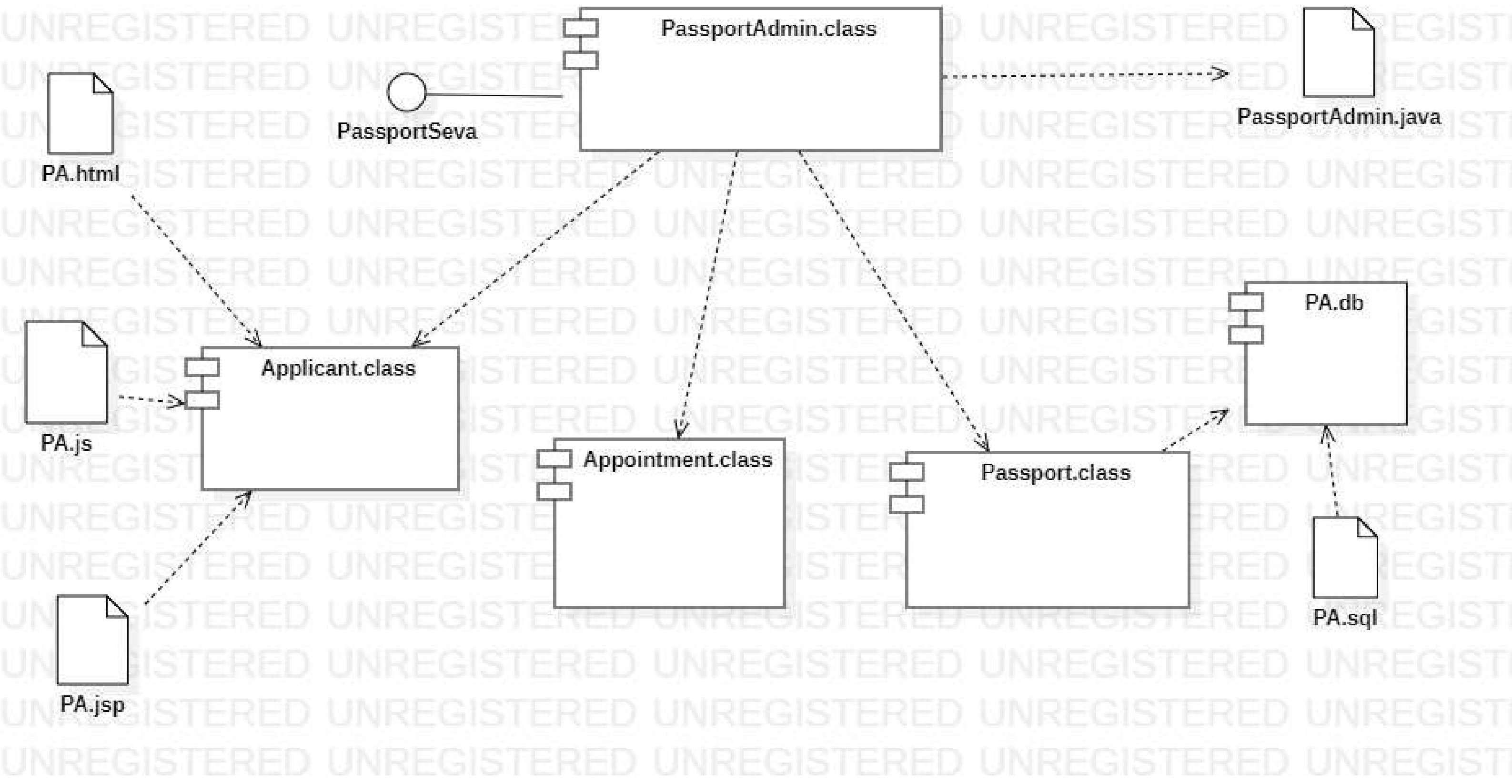
Life-line bar

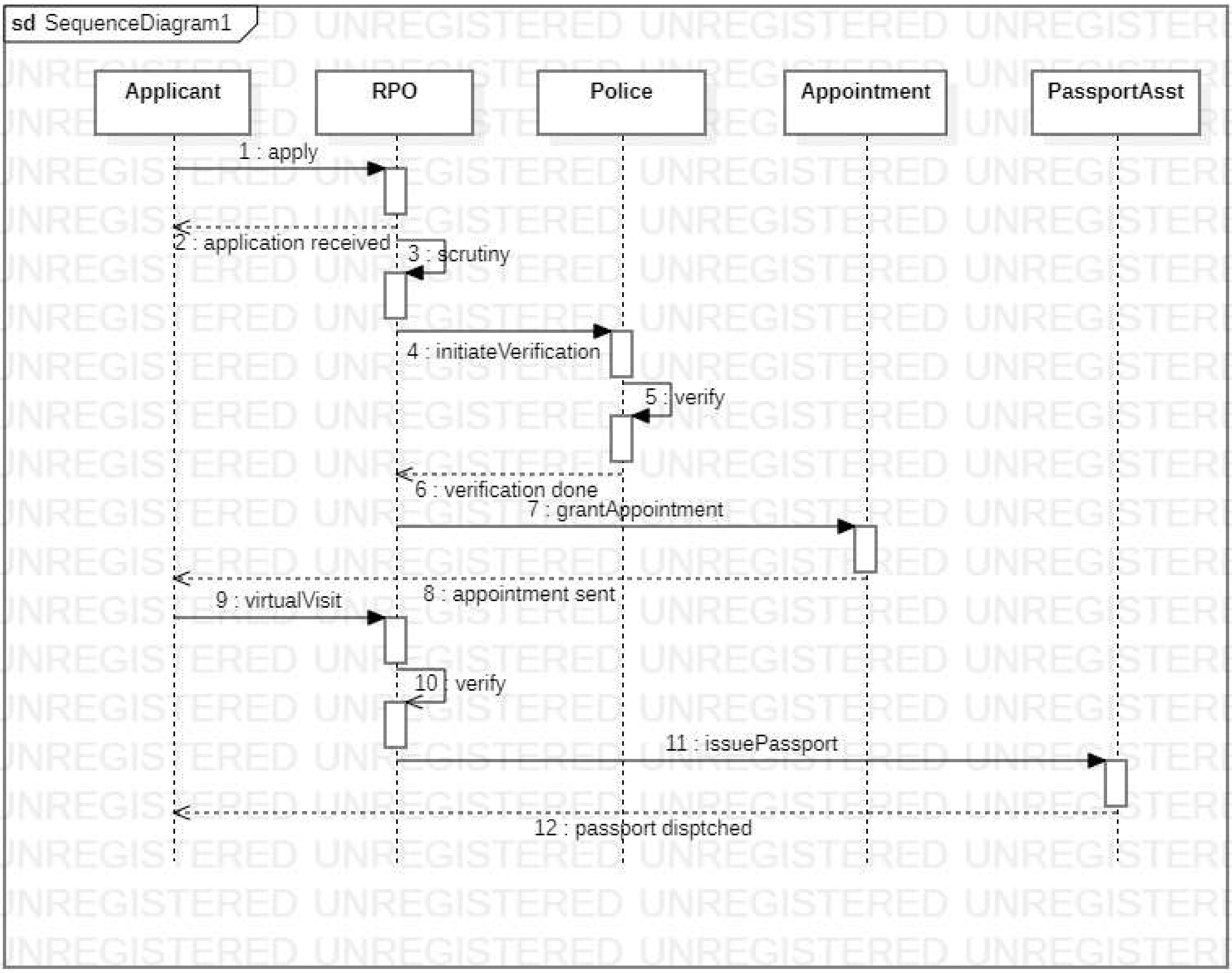
A down-ward vertical line from object-box is shown as the life-line of the object. A rectangle bar on life-line indicates that it is active at that point of time.

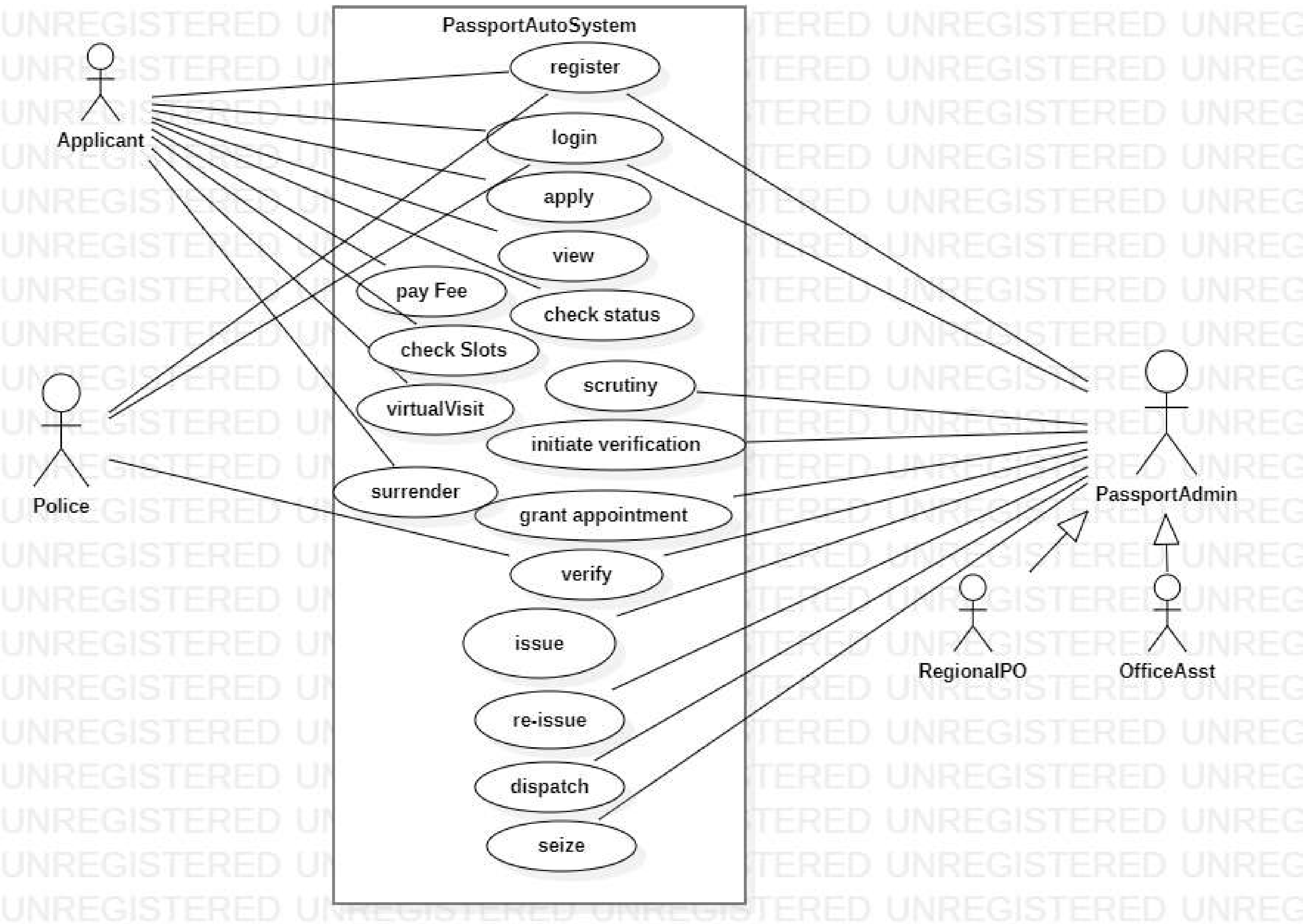
Messages

Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects' life-line show the sequence in which they occur. There may exist some different types of messages :  synchronous, asynchronous, self , reply.


Applicant
+appid: Integer {ordered, unique}
+appname: String
+typeOfService: String
+numOfPages: Integer
+requiredScheme: String
+gender: String
+dob: LocalDate
+age: String
+fname: String
+mname: String
+address: String
+mobile: String
+email: String
+country: String
+apply()
+payFee()
+checkStatus()
+view()
+virtualVisit()
+update()
+store()
+surrender()
Appointment
+apponum: Integer
+appQuota: String
+appodate: LocalDate
+appotime: LocalTime
+appoffice: String
+address: String
+appointmentsGiven: String
+appointment()
+checkAppointmentSlots()
+changeAppointment()
+display()
+store()
PassportAdmin
+officerid: Integer
+officername: String
+designation: String
+search()
+scrutiny()
+initiateVerification()
+grantAppointment()
+verify()
+store()
Police
+pid: Integer
+pname: String
+pdesignation: String
+psname: String
+verify()
+issueCertificate()
+store()
Passport
+passportnum: String {ordered, unique}
+type: String
+regionaloffice: String
+country: String
+issuedate: LocalDate
+expirydate: LocalTime
+createPassport()
+updatePassport()
+deletePassport()
RPO
+issuePassport()
+re-issuePassport()
+seizePassport()
OfficeAsst
+dispatchPassport()
Useraccount
+username: String {unique}
+password: String
+otp: Integer
+register()
+createUser()
+login()
+logout()
+resetPassword()
+blockUser()
+deleteUser()
Certificate
+dobProof: String
+addressProof: String
+aadhar: String
+addCertificate()
+viewCertificate()
+deleteCertificate()
1
1
1..*
1
1
1..*
1
1
1
1..*


PassportAdmin.class
PassportSeva
PassportAdmin.java
PA.html
PA.js
PA.jsp
Applicant.class
Appointment.class
Passport.class
PA.db
PA.sql


sd SequenceDiagram1
Applicant
RPO
Police
Appointment
PassportAsst
1 : apply
2 : application received
3 : scrutiny
4 : initiateVerification
5 : verify
6 : verification done
7 : grantAppointment
8 : appointment sent
9 : virtualVisit
10 : verify
11 : issuePassport
12 : passport disptched


PassportAutoSystem
Applicant
register
login
apply
view
pay Fee
check status
check Slots
scrutiny
virtualVisit
initiate verification
surrender
grant appointment
verify
issue
re-issue
dispatch
seize
Police
PassportAdmin
RegionalPO
OfficeAsst