TRANSIT DROID

Architecture Document

Transit Droid Team

Austin Takam
Daniel Magni
Paul Smelser
Razvan-Lada Moldovan
Yasser Al-Hasan

REVISION HISTORY

Date	Version	Description
2012-10-26	0.1	Basic infrastructure for mapping to database
2012-12-13	0.2	Domain Layer: Initial Design and revamped mapping to split Input and Output Mappers
2013-02-14	0.3	Unit of Work
2013-02-20	0.4	Proxy Pattern
2013-03-06	0.5	

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This document describes the architecture of the transit Droid system. It explains the significant architectural decisions made throughout the development process. Each revision of this document will detail the aspects relevant to the work completed during the corresponding sprint.

1.1 PURPOSE

This document is intended to provide developers with a clear understanding of the architectural patterns used in the Transit Droid system and the reasons for choosing them. This will be done by using the 4+1 model.

1.2 SCOPE

Due to a non-disclosure agreement with the primary stakeholder the *Société de transport de Montréal (STM)*, some aspects of the Transit Droid system cannot be documented here. For those sections, we will provide access to the confidential documentation only to persons who have sign the confidentiality agreement and at the discretion of the STM. The other aspects of the Transit Droid system will be documented here at a high level. For an in-depth look at the individual elements of the Transit Droid system please see the API documentation.

2 ARCHITECTURAL GOALS AND CONSTRAINTS

2.1 SECURITY

One of the many advantages of the Transit Droid software architecture is that it is JAVA Based, which is well known for its strict security without sacrificing performance, reusability, and scalability. The Transit Droid software must be secure and prevent miscellaneous users from forging tickets into the system. Transit Droid consists of two systems, the Android end and a web service end. To accomplish security, communication between these two systems is done through a set of authentication credentials consisting of user password, SALT, and login keys.

A run-able mobile and web version of Transit Droid should be available for STM clients. The Application end of the system should run on the Client side, and the application Logic (Domain Layer) will be running on an application server which is highly secure. The Database will be stored on a highly secured Data server. Our Stakeholder's servers i.e. STM are highly secured and passing their firewall is extremely difficult which will insure our source code is secure, and prevent data leakage and provide proper data privacy.

2.2 DEVELOPMENT TOOLS

The development tools and technologies selected shall be able to provide the following:

- Produce fast and rapid System Component prototypes for clients to give feedback on.
- Will provide the team the ability to take advantage of object-oriented architecture when providing a Desktop Application.
- Provide stable development process data and development services from stable companies that are not going out of business.
- Companies in charge of the development services are considered and are expected to remain market leaders.
- Insure that the development team is on the planned strategic development plan.
- Produce progress Iteration Burn-down charts, to show clients progress.
- Track defects, prove they are solved and mark them as resolved.
- Have a versioning control system which helps share changes and allow team members to work on code concurrently.
- Provide a document sharing platform for Requirement Documentation, Design and Implementation Documentation, Testing Documentation, and Team Management ex: Task Assignment.
- Allow Development team to communicate online to ease communication.
- Track Accomplished Tasks, Under Process Tasks, Future Tasks, and Immediate Task Assignment.
- Set up Presentations and Product Demos.

2.3 REUSE

As demand for application reusability is increasing, the application is developed in the form of layers and partitions. The application Logic (Domain) is going to be installed on an application server and The STM IT department will be granted access to the application source code. The use of the application with time is expected to increase the size of data stored in the database on the Data server. Adopting a Layered framework allows complex problems to be partitioned into smaller more manageable incremental strategies that are easy to reuse and adapt to changes.

Different implementations of the same layer can be used interchangeably, provided they provide the same interfaces to their adjacent layers. Lower Layers in the system contain a lot of reusable functions. Using an Object Oriented programming framework (JAVA) also helps improve reusability as Inheritance is taken into account. Inheritance allows for reusing implementation and interfaces to support an is-a relationship. Changes are also made at ancestor levels, which will be inherited in various children parts of the application.

2.4 DESIGN AND IMPLEMENTATION

The design and implementation of The Transit Droid System is done in an iterative process. With an iteration time window of 2 weeks, the development team has enough time to discuss new client needs and requirements, as well as design and implement new desired and scheduled features to the system. The skills of various team individuals differ, which allows more than one component to be designed and implemented at the same time. The Layered architecture framework in hand allows for various team members to design and implement different system components concurrently.

2.5 TEAM STRUCTURE

The development tools and technologies should allow team members with different skills to take a development role in the part they like to work on or master. This allows each team member to capitalize on the things they are strong in. The development tools will allow team members to work individually and as a team without slowing their pace. The development team is structured in a way where everyone does the job he/she does the best. The team is divided into documenters, testers, and developers according to the project needs. Team Members alternate roles to insure complete knowledge of the system and stay synced with the development pace.

3 USE CASE VIEW

3.1 PRIMARY USE CASES

The Transit Droid system is a multi-platform system, however, the core of the system must be centralized. Primarily, this is for security reasons.

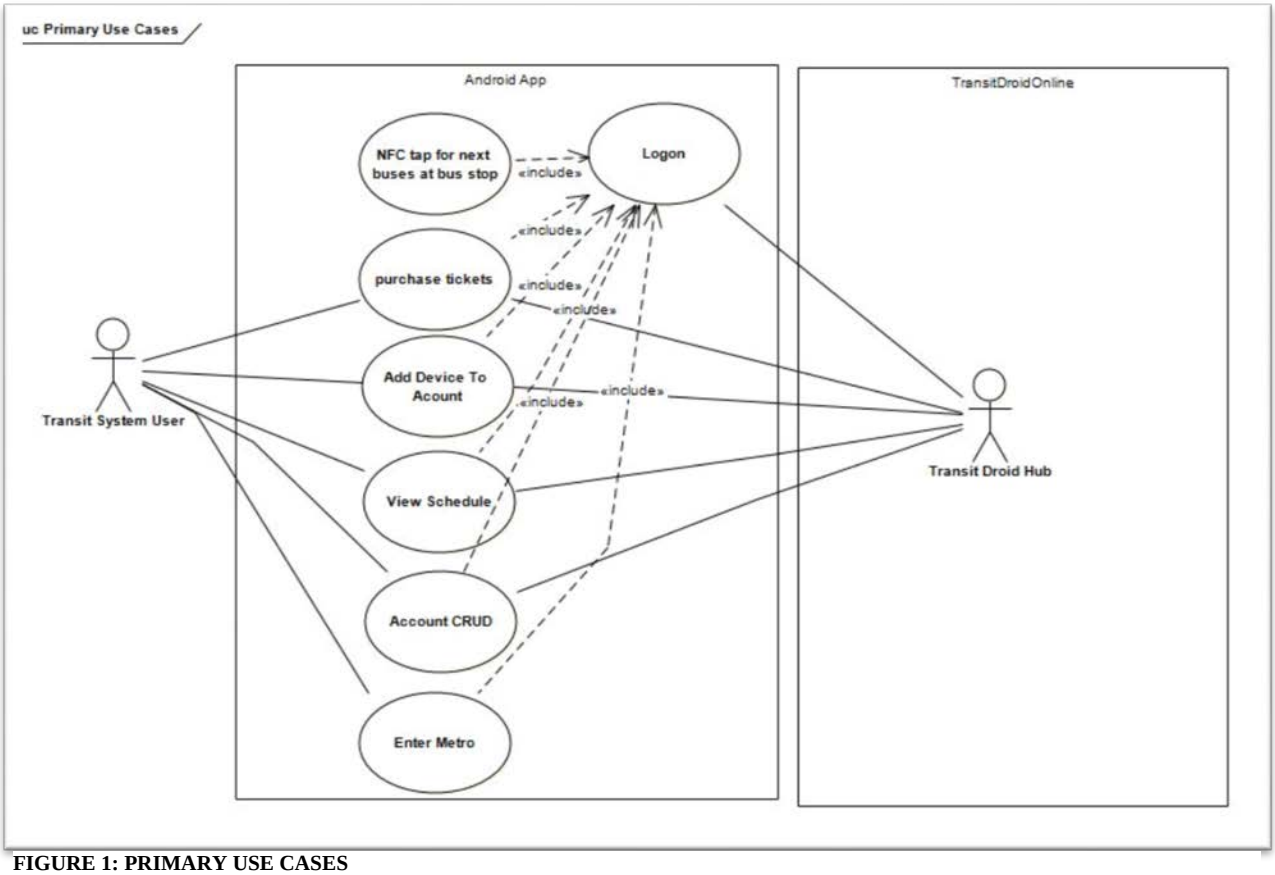

FIGURE 1: PRIMARY USE CASES

3.2 DETAILED USE CASES

4 LOGICAL VIEW

4.1 INTRODUCTION

4.2 ARCHITECTURAL STYLES

The Transit Droid system is a multi-platform system, however, the core of the system must be centralized. Primarily, this is for security reasons. When designing Transit Droid, two possibilities were considered. The first we considered keeping all data related to transit system fares on the Android device itself. In this scenario, the Transit Droid system would operate as if the Android device was a smart card and would only require an outside system to purchase fares. All information pertaining to contracts (fares) held by the client would be in the phones memory—most likely in the secure elements of the device. The second possibility is to keep minimal information on the device. The majority of the account and contract information would be kept in the Transit Droid servers. The Android device sends encrypted verification data to Transit Droid and all actions are completed on the server. Ultimately, we chose the second alternative because it provided better security.

4.2.1 SERVICE ORIENTED ARCHITECTURE (SOA)

Web services were chosen to handle requests to Transit Droid for two reasons. First, an Android application is the primary user interface for Transit Droid, therefore, system had to be remotely accessible. Second, Transit Droid is required to handle a high volume of request (1). Web services are easily scalable and are designed to receive remote requests.

4.2.2 LAYERED ARCHITECTURE

Transit Droid is built using a layered architecture. There are several reasons for this choice. The following are among the strongest.

1. Separation of Concerns: It was necessary to separate our user interface from the domain layer, in part, because the user interface is both a web application and an Android application. However, the system also had to be extensible enough to add future, unforeseen requirements, be highly maintainable and scalable. Layered architectures all lend themselves well to these requirements.
2. Localizing changes: Transit Droid is required to be adaptable to change. By using a layered system, a component of the system can be completely replaced without having a drastic effect on the rest of the system.
3. Security: By using a layered system, Transit Droid's secure layers can be hidden behind firewalls.

Transit Droid is split into Application, Service, Domain, Technical, Data, and two Presentation layers (Android and Web).

4.2.2.1 *Presentation Layer*

Consists of both the Android application and a web application.

4.2.2.2 *Application Layer*

Has actions that take incoming HTTP requests from the web service, creates the appropriate 'Command' and returns the results to the correct page in the Presentation layer.

4.2.2.3 *Service Layer*

Handles RESTful web service calls. Any action that requires access to the core system uses a web service call.

4.2.2.4 *Domain Layer*

The domain layer is broke down into three main sections, core, entities and logic. The core section of the domain layer has all the base level domain classes and interfaces that are used by the rest of the system.

4.2.2.5 *Technical Layer*

Any special services required by the system. This includes encryption service and a registry used to access the system properties.

4.2.2.6 *Data Layer*

Acts as a gateway to the database. All queries to the database are done using Table Data Gateways.

4.3 Domain Model

4.3.1 Main Elements

In this section, the main domain elements will be explained at a high level. Transit Droid must work with both smart cards and mobile devices (reference req here). Therefore, the domain was designed so that the mobile device operates in *smartcard simulation* mode. In the initial design—seen in Figure 1: Initial Design—the Mobile Device was treated as if it was a smartcard itself.

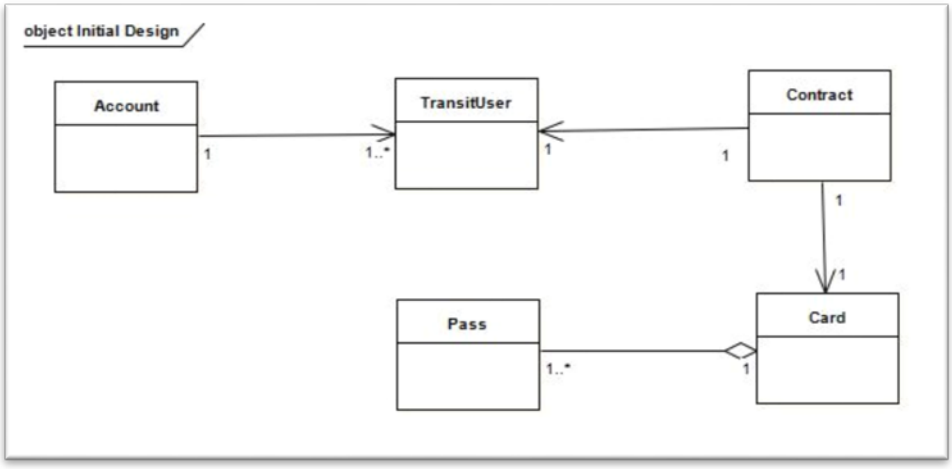

FIGURE 2: INITIAL DESIGN

The second design for Transit Droid extracted the Mobile Device. This was done for two reasons. First, it is a closer representation of the *real* domain for the Transit Droid system. A mobile device is used as part of the system and the systems had to have some way to keep track of its mobile devices. Second, for extensibility, it gives the system added flexibility to have the mobile device as a separate entity.

In Figure 2: Main Domain Elements, the main elements of the Transit Droid system are shown.

- Account: Allows users to be linked together. This was done so that a family can be under one account. This facilitates (req), which requires the system to allow a member of a family to purchase tickets for others in the family.
- TransitUser: A person who rides the transit system.
- Contract: Links TransitUsers with their Card data and mobile devices. The contract satisfies (req) by allowing a TransitUser to be linked to multiple Cards and/or multiple MobileDevices.
- MobileDevice: Am Android phone or tablet.
- Pass: Contains fare details. Six types of transit fares are required (req) in Transit Droid, Daily, Monthly, Nightly, Yearly, Three Day, and Single.
- Card: Has multiple passes. Each pass may have valid fares available. (req – multiple fares charged at same time – logic to know which to use)

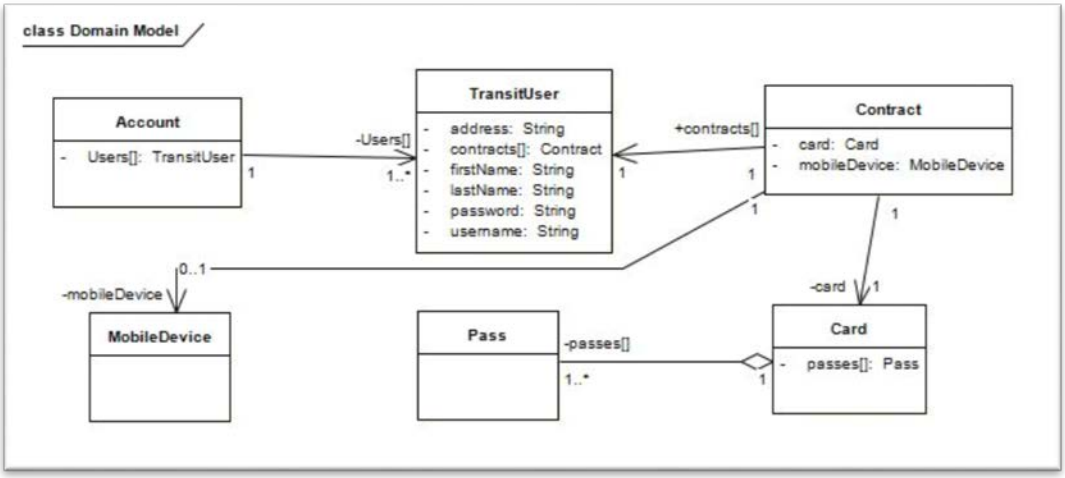

FIGURE 3: MAIN DOMAIN ELEMENTS

4.3.2 PACKAGE MODEL

Figure 3: Package dependency, shows all layers of the system, except the presentation layer. Transit Droid follows a strict layered architecture. Layers are always only dependent on the layer directly below them. This gives the system greater flexibility for change as is required in (req).

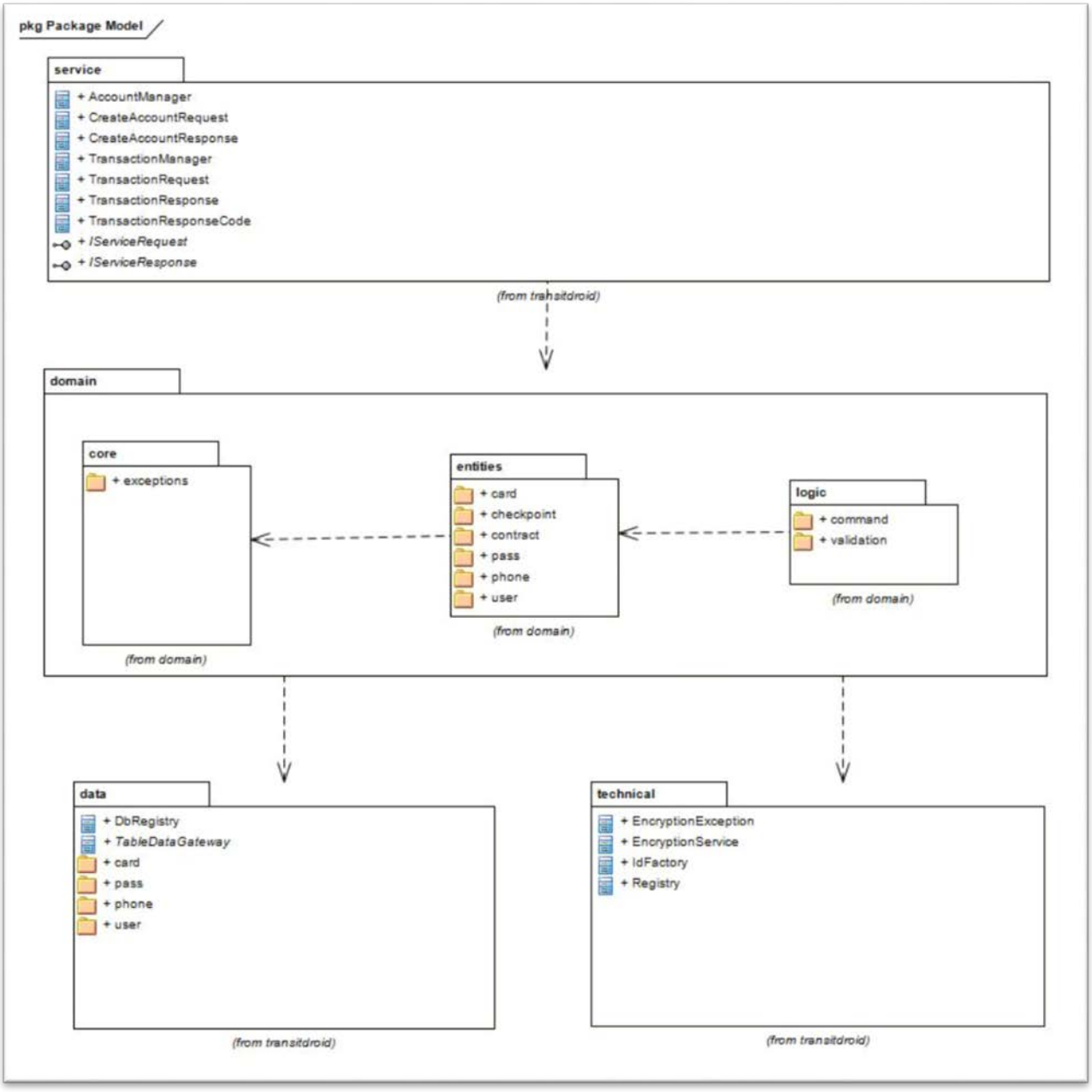

FIGURE 4: PACKAGE DEPENDENCY

4.4 CLASS MODEL

The class model will give a more in depth look at Transit Droid both using static and behavioural UML diagrams and brief descriptions. The main elements of the domain layer will be described first, followed by the domain core, the data layer and finally any architecturally significant patterns used.

4.4.1 DOMAIN LAYER AT A GLANCE

The domain layer is broken into three packages—Core, Entities and Logic. The Core package is a collection of interfaces and abstract classes that define the domain structure and functionality. The Core will be explained in detail in section 3.4.2. The entities package contains the main domain classes first introduced in the domain model—Account, TransitUser (User), Contract, Card, MobileDevice, and Pass.

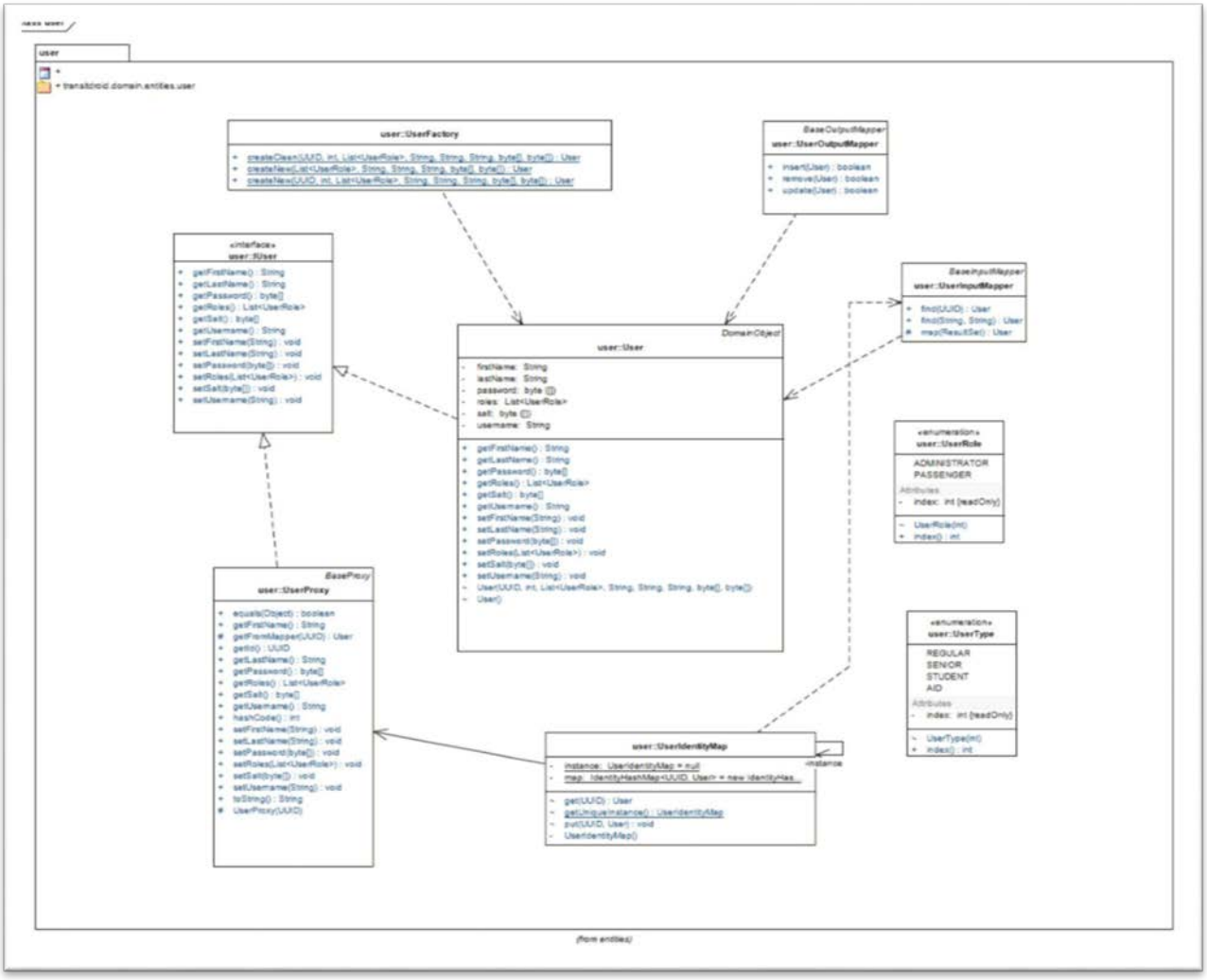

FIGURE 5: USER PACKAGE

Mappers, Factories, Proxies and etcetera related to each of the main domain entities are contained in the same package. This facilitates information hiding without compromising the functionality of the system. *Figure 4: User package* shows the domain.entities.user package.

4.4.2 DOMAIN CORE PACKAGE

The domain.core package contains interfaces and abstract classes that define the domain layers framework. All classes of the domain layer inherit from or use these base classes and interfaces.

4.4.2.1 IDomainObject

Attribute	*Type*	*Description*
Id	UUID	Id used as the primary key for persistent storage in relational tables
Version	Int	Used to manage concurrency concerns like lost updates

FIGURE 6: IDOMAINOBJECT TABLE

4.4.2.2 DomainObject

Method	*Return Type*	*Description*
Constructor	DomainObject	Defines the parameterized constructor with the id, and version fields

FIGURE 7: DOMAINOBJECT TABLE

4.4.2.3 BaseInputMapper and BaseOutputMapper

In Transit Droid, the mapping from the data layer to domain layer is broken into input and output mappers. The BaseInput and Output mappers have abstract methods for retrieving and putting data to the data layer.

4.4.2.4 MapperRepository

Developed to manage InputMappers and OutputMappers. Instead of creating an InputMapper and an OutputMapper for each DomainObject, the MapperRepository dynamically creates singleton input and output mappers for any type of DomainObject. When a mapper is needed again, the mapper created previously is reused.

4.4.2.5 UnitOfWork

If data is changed in-memory, it needs to be persisted to the data source for other request to see it. If not done systematically, keeping track of what is changed is difficult as a system becomes larger. Alternatively, writing out changes frequently can be slow and impractical; there may be many changes, and an actual transaction opened in a data source may need to persist across multiple requests [Fowler 2003, p.184].

4.4.2.6 BaseProxy

Defines the basic functionality for DomainObject proxies.

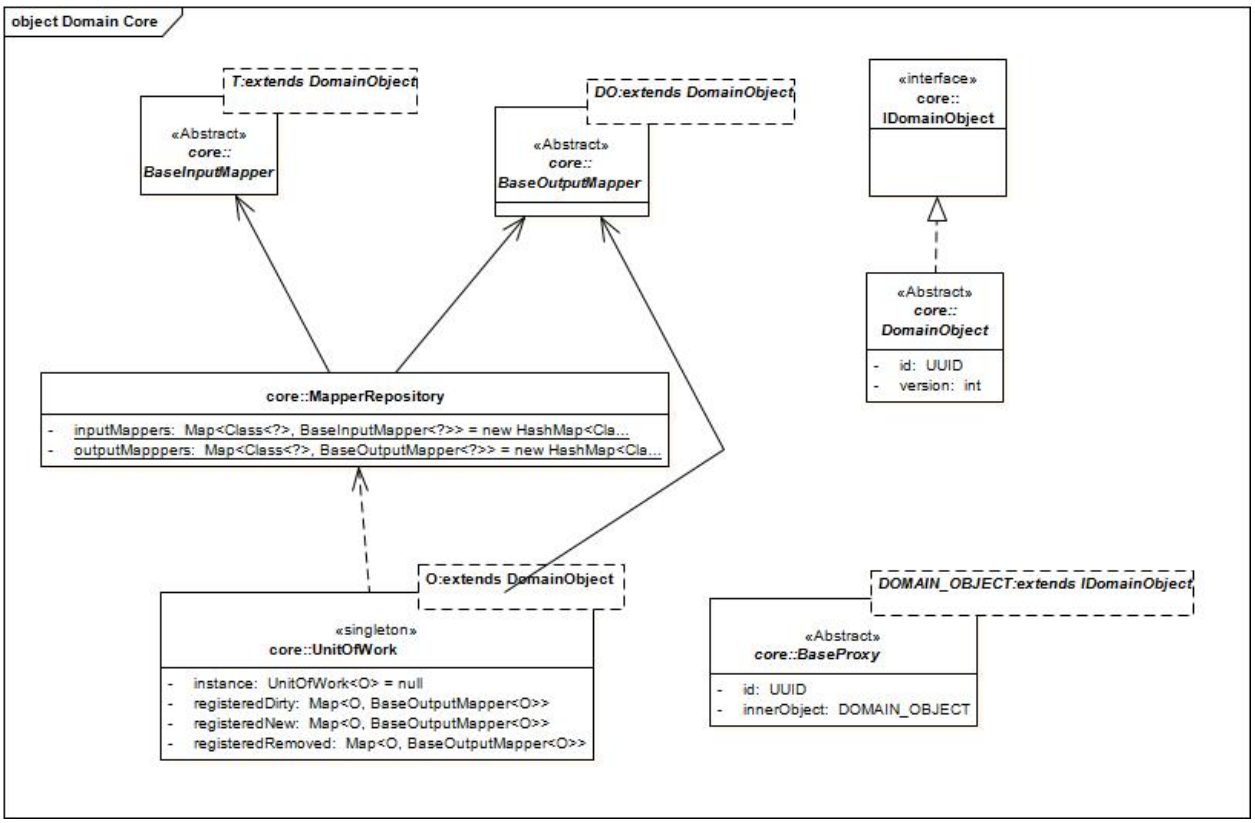

FIGURE 8: DOMAIN CORE

4.4.3 MAPPING TO THE DATABASE

Mapping to and from the database is done using an *Input Mapper* and *Output Mapper* for each domain object with a corresponding *Table Output Gateways* (TOGs) and *Table Input Gateways* (TIGs). The *Input Mappers* call functions from the *Table Data Gateways* (TDGs) that return Result Sets and map them to domain objects. The Output Mappers take domain objects and convert so that they can be sent to methods in the TDGs.

The separation of the *Mappers* and TDG's into "Input" and "Output" subtypes allows for better separation of concerns. In the Transit Droid system, the TDG has methods to handle setting and executing prepared statements. Each method of the TIGs and the TOGs must be declare as 'synchronized' to ensure the accuracy of incoming and outgoing data.

4.4.3.1 MAPPER CLASS DIAGRAM:

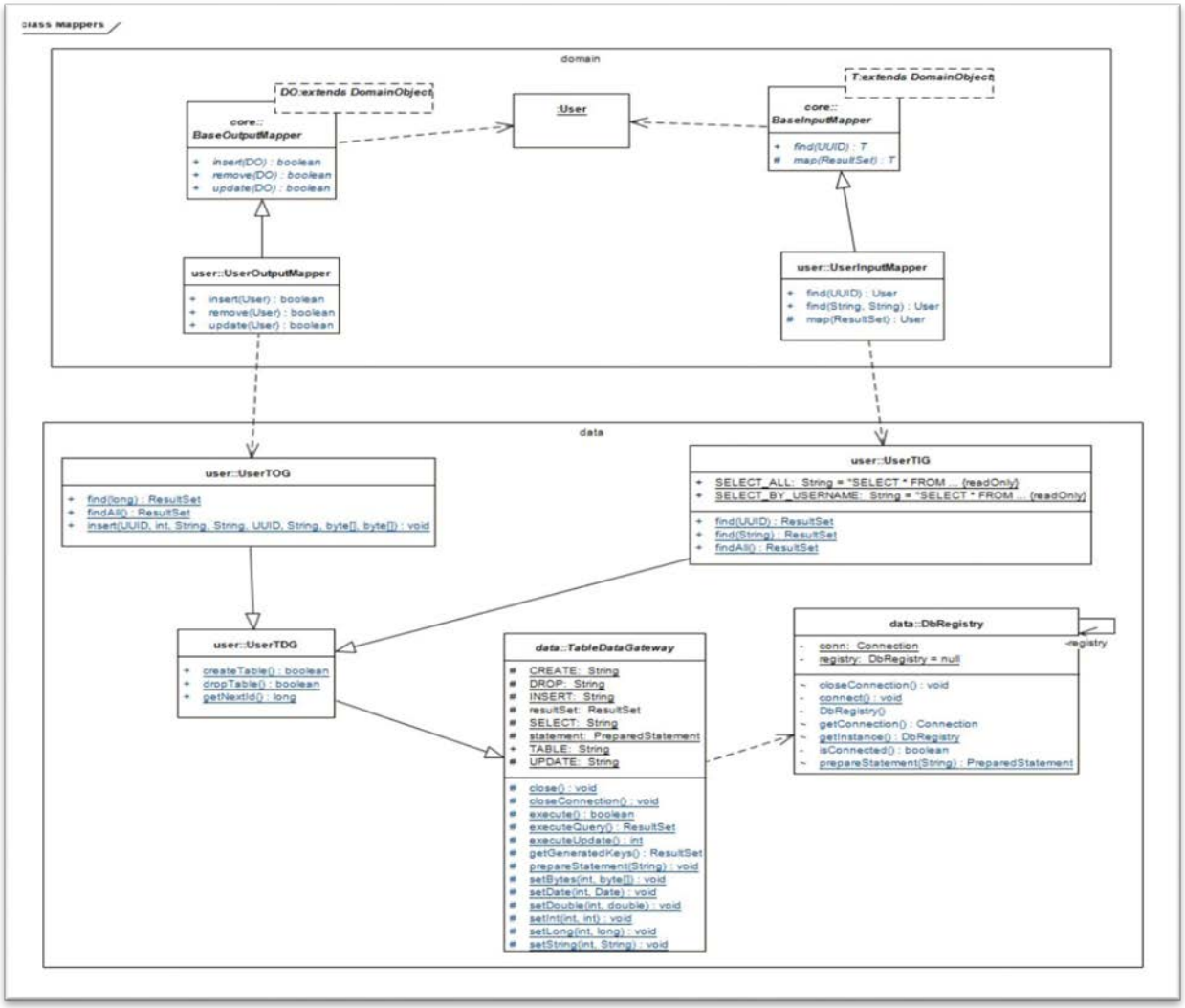

FIGURE 9: MAPPERS

4.4.3.2 MAPPER SEQUENCE DIAGRAM:

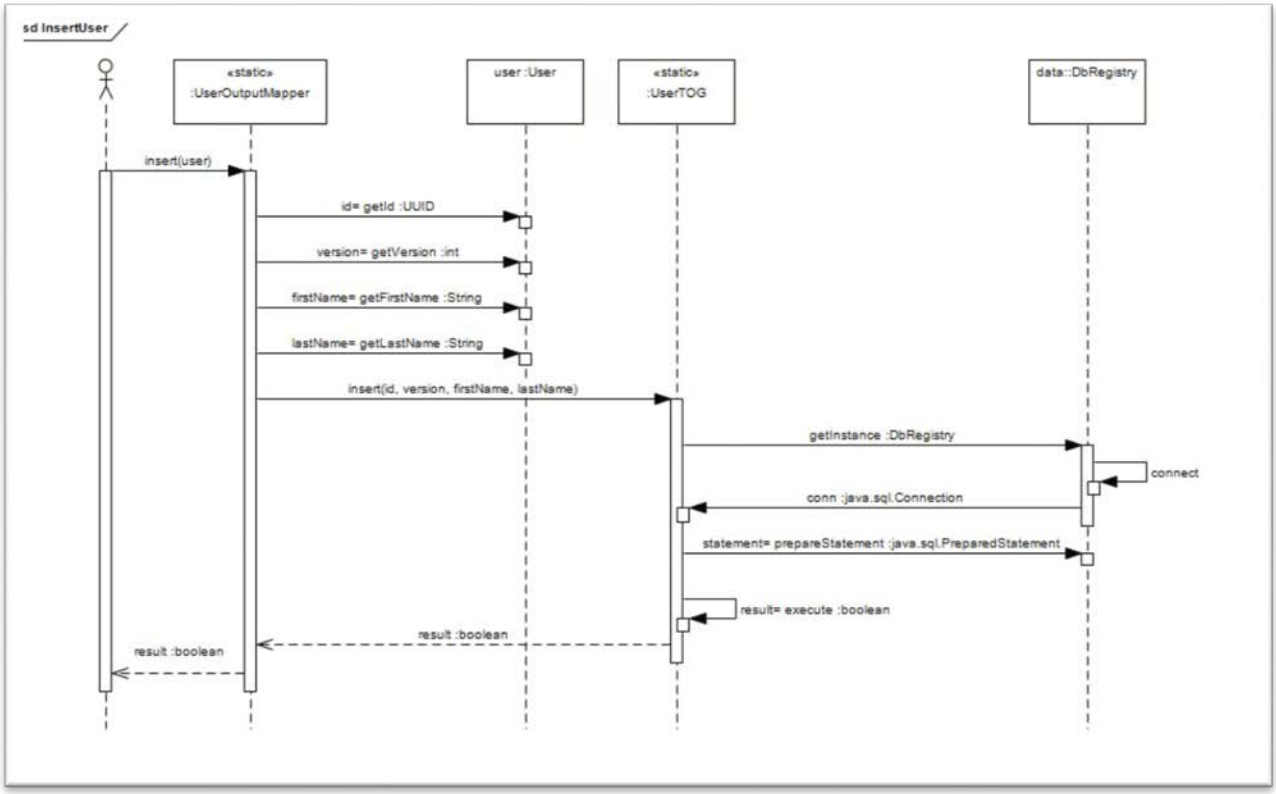

FIGURE 10: INSERT USER

4.4.4 UNIT OF WORK

The '*Unit of Work*' pattern is used to manage transactions to the database. All items effected during the course of a transaction are added to the *Unit of Work* and categorized as dirty, new or removed. The Transit Droid *Unit of Work* retrieves an *Output Mapper* from the *Mapper Repository* and calls the appropriate method to make the necessary changes.

4.4.4.1 UNIT OF WORK CLASS DIAGRAM:

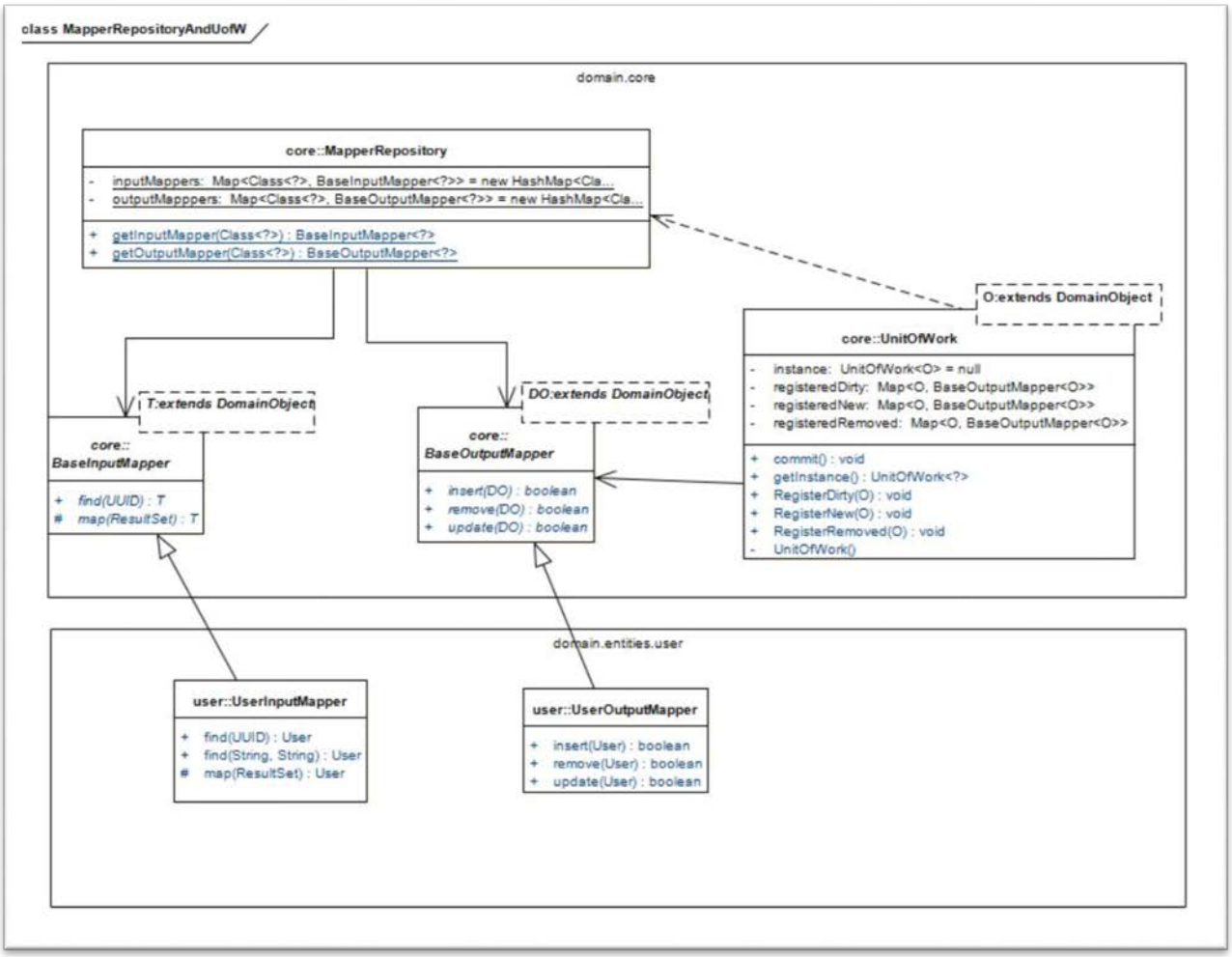

FIGURE 11: UNIT OF WORK

4.4.4.2 *Unit of Work Sequence Diagram:*

This is a trivial example of the unit of work, however, it demonstrates how objects are registered with the unit of work and how the unit of work handles updates.

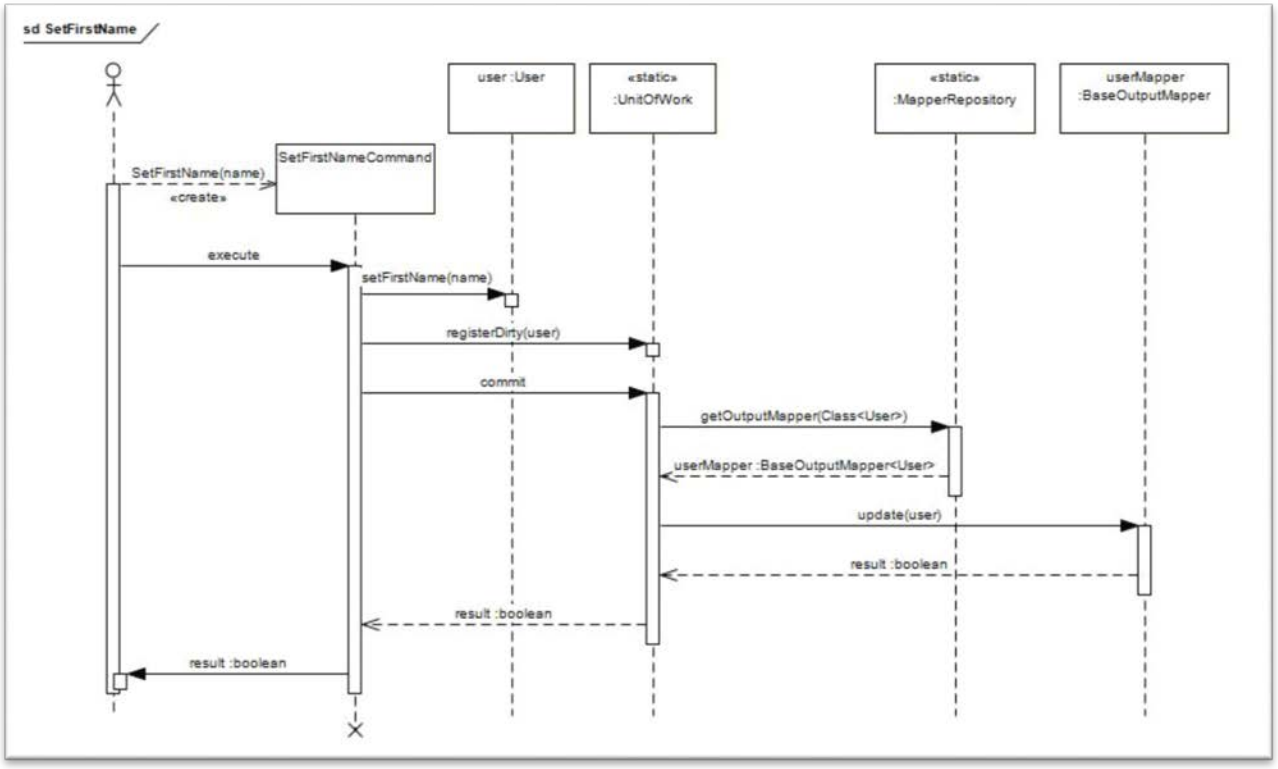

FIGURE 12: SET FIRST NAME

4.4.5 Resource Conservation – Proxies

Because Transit Droid will have to handle large volumes of online traffic, it is important to minimize calls to data and memory required to handle transactions. For this reason, lazy load proxies are implemented for all domain objects. Whenever an object is loaded from the data layer, any composite objects are created as empty proxies. Only if the details of the real object are required will the data be retrieved from the data layer.

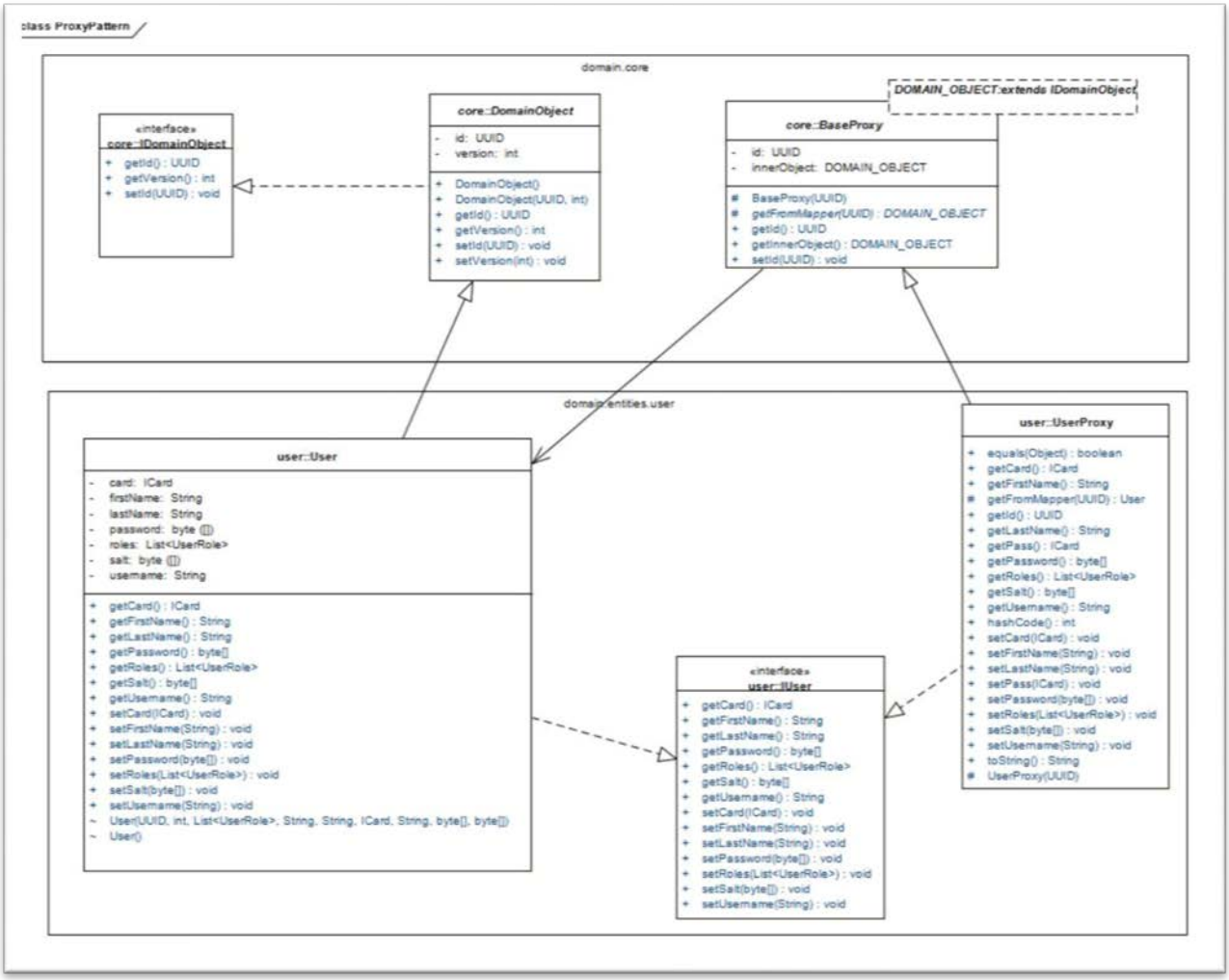

FIGURE 13: PROXIES

5 PROCESS VIEW

5.1 INTRODUCTION

6 DEVELOPMENT VIEW

6.1 INTRODUCTION

7 PHYSICAL VIEW

7.1 INTRODUCTION