

# System Design: Parking Garage Automation

Matt Edwards, Eric Wasserman,  
Abdul Hassan, Juan Antialon, Luke Steepy

Group Number 2

Project URL: <http://code.google.com/p/parking-garage-automation/>  
<http://www.park-a-lot.vacau.com/>

## Effort Estimation

The table below is a breakdown of the individual member's contributions to each part of this report

	<b>Abdul</b>	<b>Matt</b>	<b>Eric</b>	<b>Luke</b>	<b>Juan</b>
Interaction Diagrams (35 points)		50%	50%		
Classes and Interface Specifications (13 points)		40%		60%	
System Architecture and Design (22 points)	25%		10%		65%
Algorithms and Data Structures (4 points)	40%	40%			20%
User Interface (10 points)	100%				
Plan of Work (4 points)		50%			50%
References (2 points)					100%
Project Management (10 points)		60%	40%		
<b>Total Points Contribution</b>	17.1	32.3	23.7	7.8	19.1

# Table of Contents

<b>EFFORT ESTIMATION .....</b>	<b>2</b>
<b>INTERACTION DIAGRAMS .....</b>	<b>4</b>
NOTES AND CONVENTIONS .....	4
UC-1: RESERVE .....	5
UC-2: PARK .....	8
UC-3: MANAGE ACCOUNT .....	15
UC-4: MANAGE RESERVATIONS .....	18
UC-5: REGISTER .....	20
UC-6: MANAGE GARAGE .....	23
UC-7: CANCEL RESERVATION .....	25
UC-8: EXTEND RESERVATION .....	28
UC-9: AUTHENTICATE USER .....	31
UC-10: SET PRICES .....	33
UC-11: INSPECT USAGE HISTORY .....	35
<b>CLASS DIAGRAM AND INTERFACE SPECIFICATION .....</b>	<b>37</b>
CLASS DIAGRAMS .....	37
DATA TYPES AND OPERATION SIGNATURES .....	41
<b>SYSTEM ARCHITECTURE AND SYSTEM DESIGN.....</b>	<b>47</b>
ARCHITECTURAL STYLES .....	47
IDENTIFYING SUBSYSTEMS.....	49
MAPPING SUBSYSTEMS TO HARDWARE .....	51
PERSISTENT DATA STORAGE .....	51
NETWORK PROTOCOL .....	65
GLOBAL CONTROL FLOW .....	65
HARDWARE REQUIREMENTS.....	65
<b>ALGORITHMS AND DATA STRUCTURES.....</b>	<b>66</b>
ALGORITHMS.....	66
DATA STRUCTURES .....	67
<b>USER INTERFACE DESIGN AND IMPLEMENTATION.....</b>	<b>68</b>
<b>PROGRESS REPORT AND PLAN OF WORK.....</b>	<b>69</b>
PROGRESS REPORT .....	69
PLAN OF WORK .....	70
BREAKDOWN OF RESPONSIBILITIES .....	72
<b>REFERENCES.....</b>	<b>73</b>

# Interaction Diagrams

## Notes and Conventions

The following is a list of conventions used for function definitions in the system interaction diagrams below. Although these are not formal function definitions, they will help the reader to understand what a function call is trying to accomplish.

Function Call	Description
<code>updatePage( "" )</code>	The string contained in this method is a place-holder currently. Eventually, it will be replaced with an enumeration of codes, depending on what the page is to be updated to.
<code>notify( "" )</code>	Just as with <code>updatePage( )</code> , <code>notify( )</code> also contains textual descriptors that refer to an enumeration, although it is not currently well defined.
SQL Queries	Between Database Proxy and Database there are mock SQL queries, which although they are not complete, provide a rough idea of the query we would be using in our system.

## UC-1: Reserve

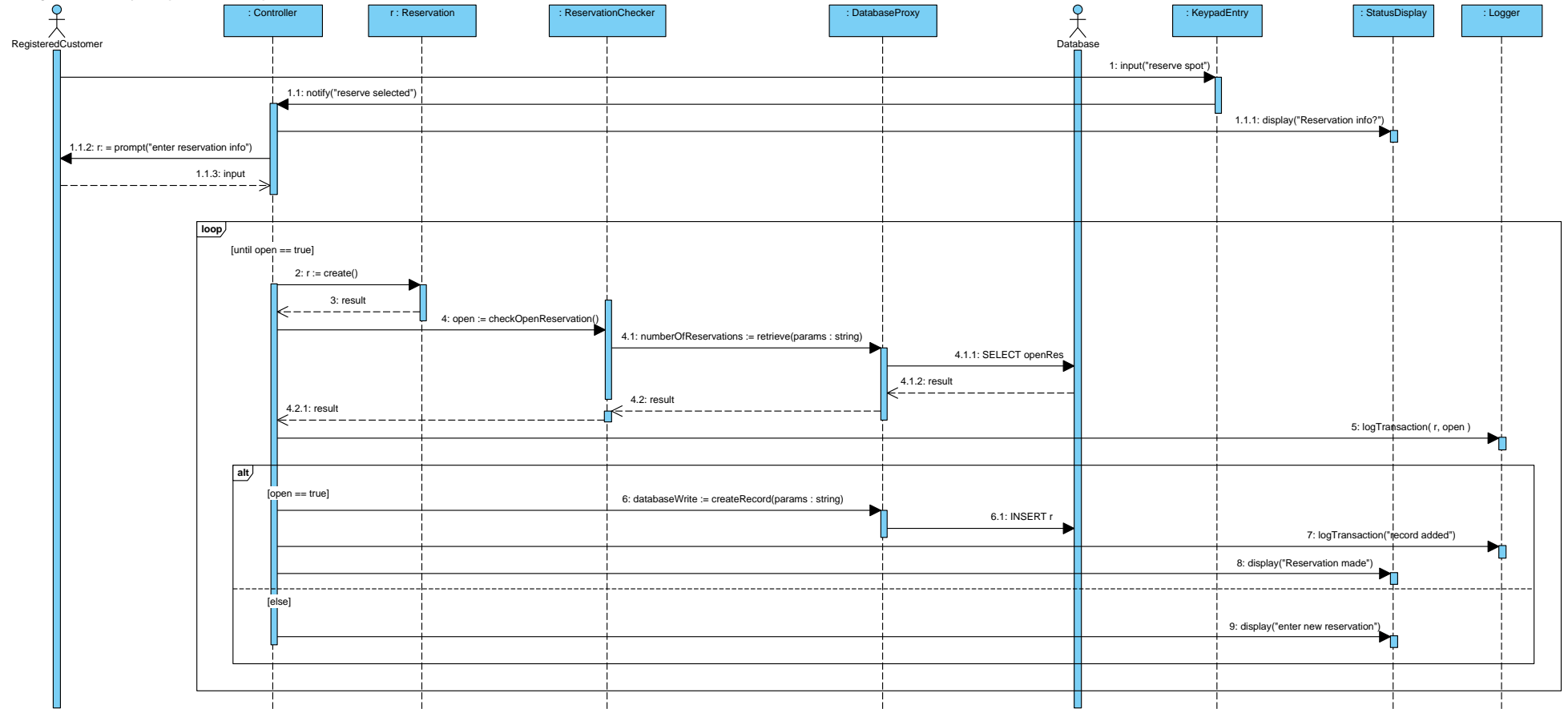
**Goals:** To create a reservation and add it to the Database.

**Process:** The customer is prompted to input information about the reservation they wish to create, then the Controller passes the information to the Reservation Checker, which determines if the reservation is valid and does not overbook the garage. The customer is then given either a success page or asked to re-enter the information with different values.

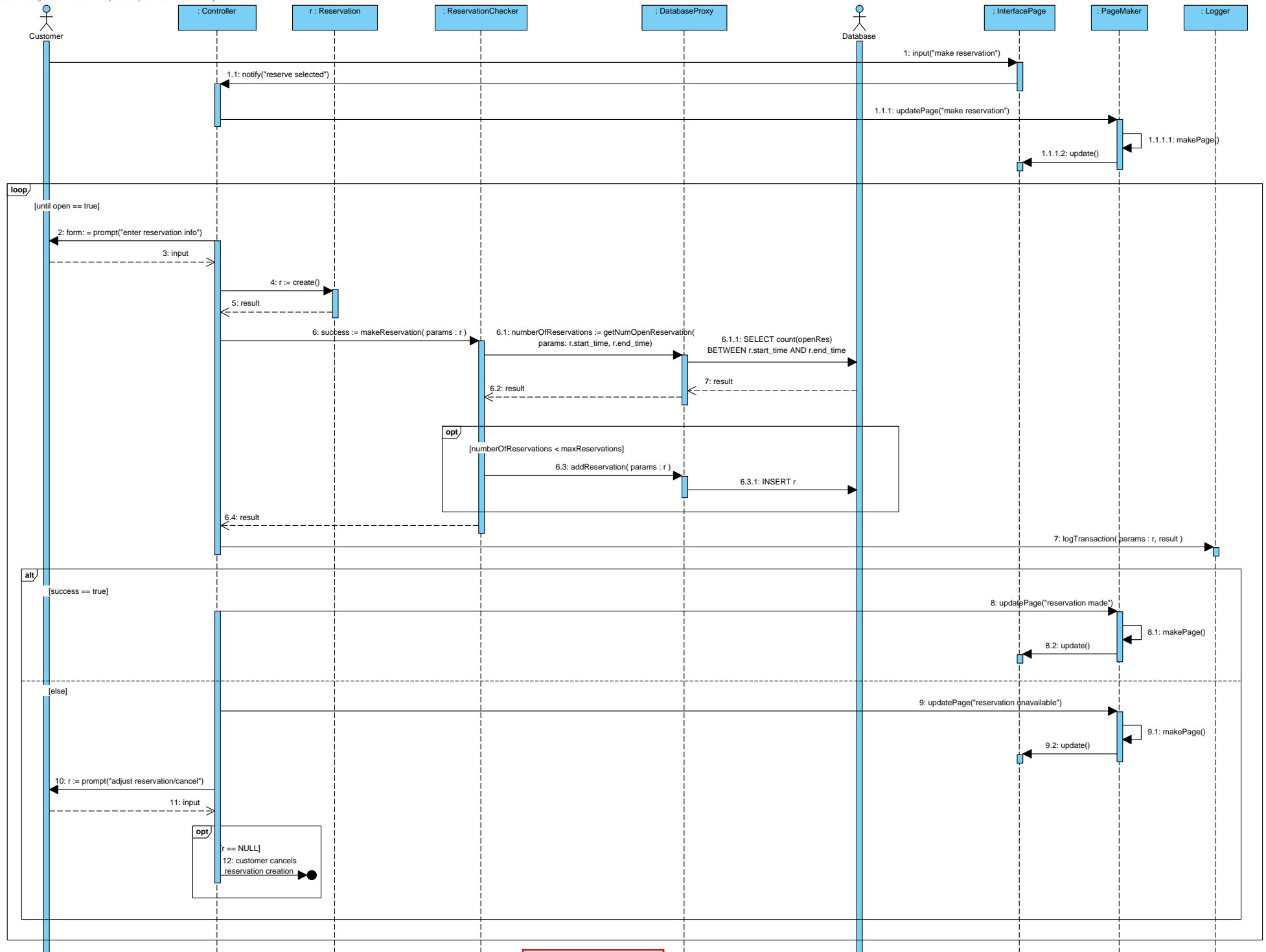
Reasons for diagram selection:

- **Expert Doer Principle** - In an attempt to keep as much data encapsulated within each class as possible, our selection of this diagram aligns well with the expert doer principle. The main acting classes are Controller, which coordinates events between the Registered Customer and the Reservation Checker, which validates an input reservation and, if possible, creates that reservation in the database. The Controller passes all of the reservation info directly to the Reservation Checker, which performs all of the logical functions on it. This division of labor ensures that the class the knows is the class performing the operation.
- **High Cohesion Principle** - In an effort to bring as much functionality to the fewest classes possible, we have only two main actors in this use case: Reservation Checker and the Controller. The Controller coordinates and the Reservation Checker performs all the necessary logic for creating the reservation. Each class has quite a few responsibilities, and therefore satisfies the high cohesion we have sought.
- **Low Coupling Principle** - Due to the small number of classes we used to implement this use case and the major responsibilities assigned to each one, low coupling follows quickly. For example, the Controller never interacts directly with the Database, rather relying on the Reservation Checker to handle that step when ordering a reservation created. This reduces the number of dependencies between our classes, creating a more flexible system.

Below are two iterations of the diagrams, where we chose the second one. This is to demonstrate our thought process and the evolution of our design.



UC-1 Reserve (deprecated)



## UC-2: Park

**Goals:** To park in the garage, either to fulfill a reservation or as a walk in.

**Process:** A customer enters the garage and is either determined to be a registered customer by having a license plate read or entering a customer ID, or parks as a walk in by making a reservation on the spot. This is a complicated use case, however, the general flow starts with a customer entering the garage elevator (Customer in Elevator) and progresses to determining the customer and reservation type (Select Reservation - Registered Customer / Walk In), then finally to the physical act of parking the car in the garage (Park),

Because of the complexity involved in use case 2 (UC-2), we have elected to break it into four (4) system interaction diagrams, which are connected to each other logically. This allows us to create more readable diagrams, each with a specific and well-defined purpose.

The diagrams are, in order of activation:

1. Customer in Elevator
2. Select Reservation
  - a. Registered Customer
  - b. Walk In
3. Park

### Customer in Elevator

This diagram covers from the Customer entering the garage elevator up until it is determined if he or she is a registered or unregistered Customer. At that point, the system interaction diagram branches.

Reasons for diagram selection:

- **Expert Doer Principle** - Here, the Controller has most of the work, since it needs to coordinate between the Elevator Camera Operator and the Database to attempt to read the license plate of the car in the elevator. Therefore, it is a bit difficult to say that this diagram satisfies the Expert Doer principle, mainly because the Controller is the one doing most of the work. However, this diagram is merely a gateway to the more complex Select Reservation diagrams. Therefore, this can be overlooked.
- **High Cohesion Principle** - The responsibilities assigned to each class are small in this diagram, although Controller has a lot of work to do coordinating data between the Elevator Camera Operator and the Database Proxy in order to determine if the Customer is registered or not. However, these responsibilities are mainly about ferrying data, and a majority of what the Controller does it log transactions into the Database for future review. Therefore, we have assigned a large number of responsibilities to each class appropriately.
- **Low Coupling Principle** - In this diagram it is difficult to say that there is low coupling, mostly because the Controller coordinates between many of the classes. However, this



is a necessary trade-off to achieve efficient operation of the system. The alternative would have been to assign more responsibility to the Database Proxy, and it would be nonsensical to have the Database Proxy talking directly to the Elevator Camera Operator, since there is a very weak relation there. Also, how would that data then get back to the Controller? Clearly, it is best to keep the Controller well connected in this case.

### Registered Customer

This diagram picks up where the previous Customer in Elevator left off. It assumes that there is a Registered Customer in the elevator and will attempt to have the Registered Customer select a reservation and begin parking.

Reasons for diagram selection:

- **Expert Doer Principle** - As before, most of the responsibility is spread between the Controller and the Key Checker (before it was the Reservation Checker). The Controller here coordinates the Registered Customer logging into his or her account, and relies mainly on the Key Checker class to perform all necessary logic to ensure that the Registered Customer enters the correct key. Therefore, the class that knows is doing the action (Key Checker checking the Key).
- **High Cohesion Principle** - There is high cohesion here because we have assigned multiple responsibilities to each class, specifically Controller and Key Checker. Controller is, as always, responsible for maintaining order and keeping the process running, while Key Checker handles the validation of the inputs Controller passes to it from the Registered Customer. This design allows for each class to have a large number of responsibilities.
- **Low Coupling Principle** - However, despite these many responsibilities, each class does not heavily depend upon each other. For instance, Controller simply asks Key Checker to verify a key and return a simple “yes/no” response, if the key is valid or not. All of the logic resides inside of Key Checker, making this diagram have low coupling and allowing for easy logic changes in the future without extensive system redesign.

### Walk In

Much simpler than the previous case since we do not need to fetch any existing reservations from the Database. Rather, this diagram assumes that the Customer in the elevator is unregistered, and is making a walk in reservation. The diagram includes a call to UC-1 Reserve to create the reservation in the garage elevator using the display and keypad.

Since this diagram relies heavily on UC-1 Reserve, there isn't much to be said about the delegation of responsibilities, expert doer principle, or coupling and cohesion. It all relies on how well UC-1 is put together.

This code re-use is great, since it simplifies our diagram and our coding later on.

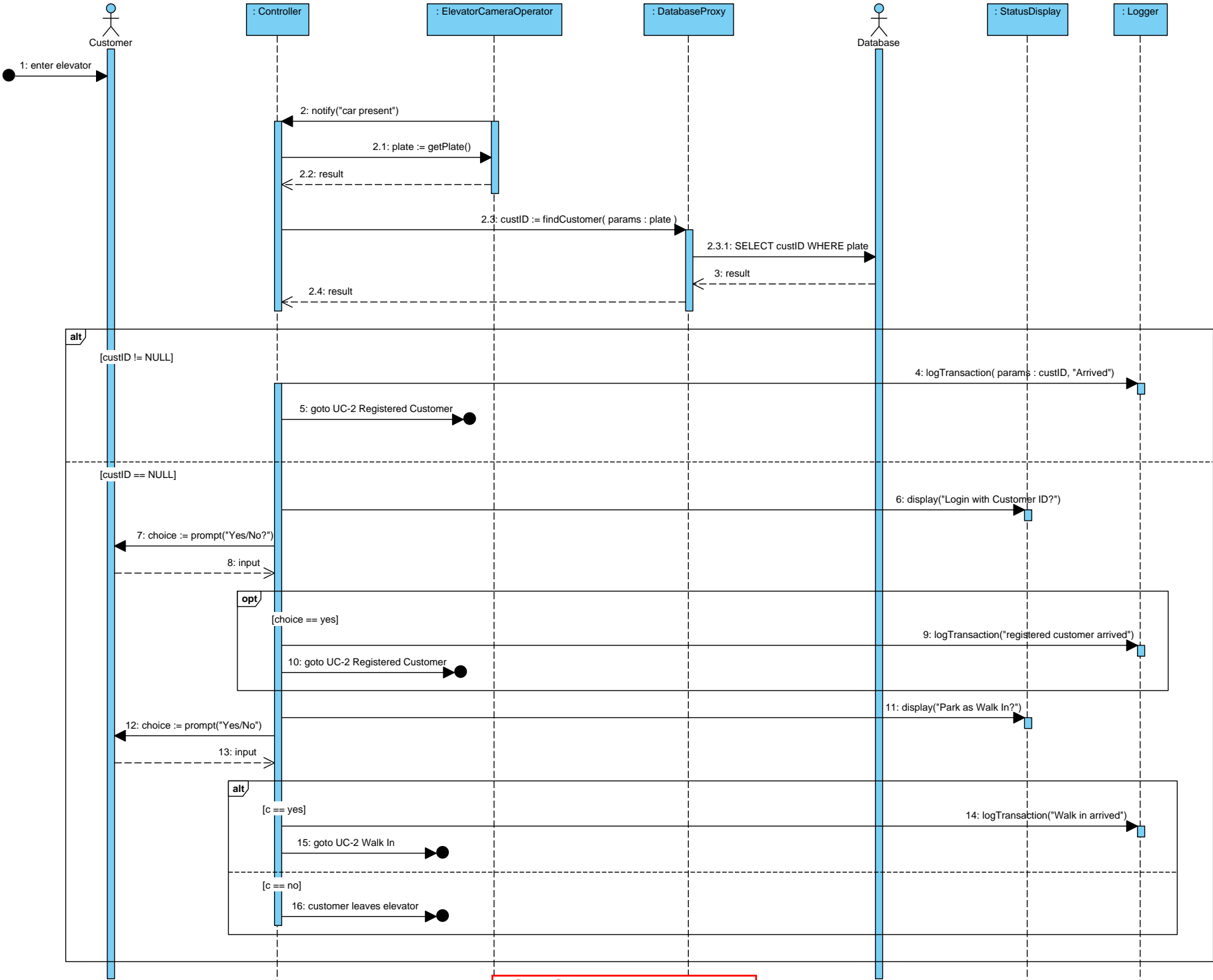
## Park

Finally, this system interaction diagram shows how a Customer will park, including moving the elevator to the appropriate floor and detecting when the vehicle is parked in a spot, up to the time it departs.

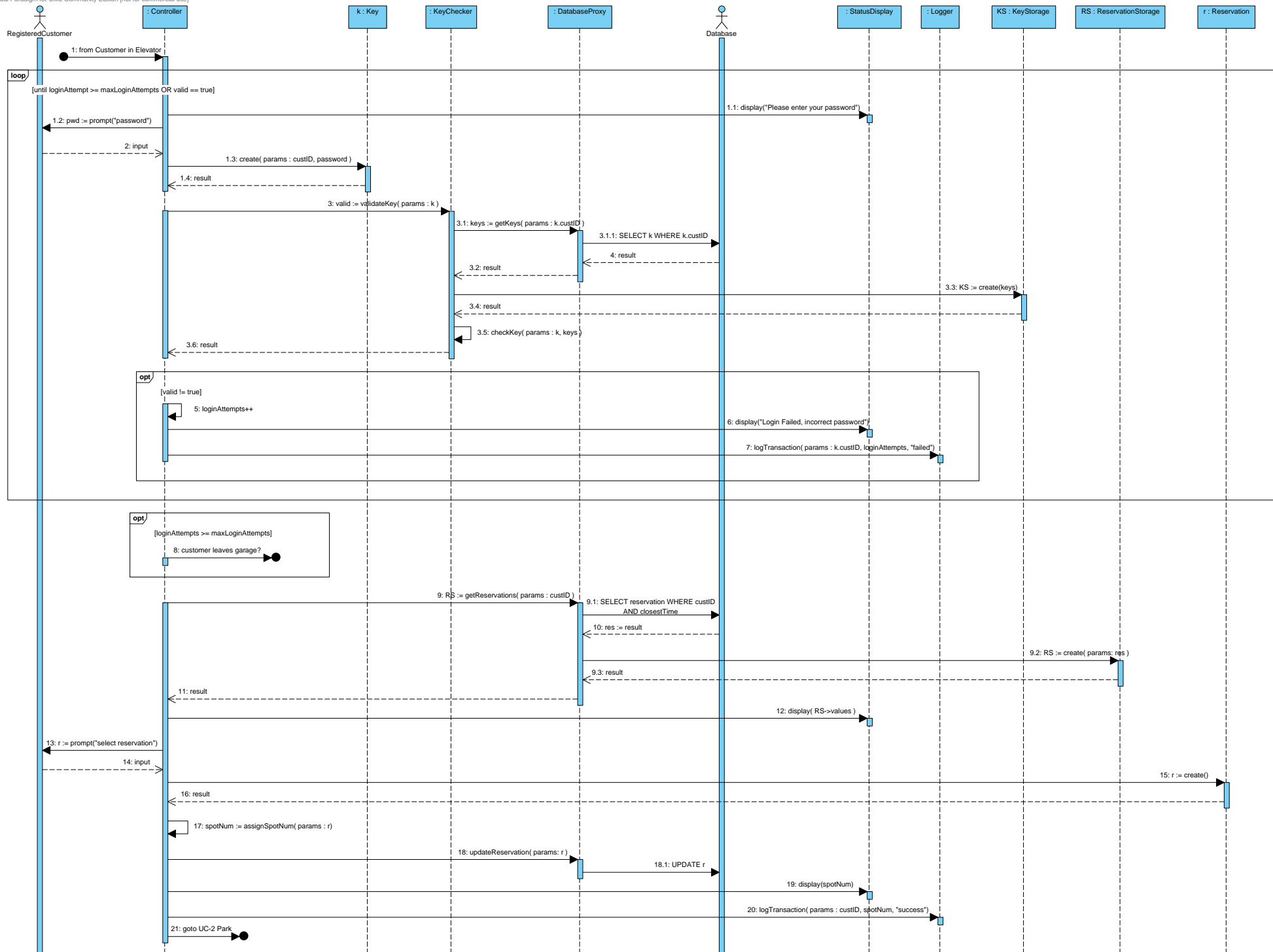
Reasons for diagram selection:

- **Expert Doer Principle** - Clearly it is easy to see how the Controller, which coordinates between the multiple systems in the garage, is the one best qualified to control the sequence of events. The Controller knows the position and location of each system operating in the garage, and can control the elevator, spot sensors, etc. to accomplish the task of parking.
- **High Cohesion Principle** - As with the Expert Doer principle, there is clearly a lot of cohesion in this design, since it relies mainly on the Controller to perform all of the functions. The remainder of the classes simply provide information to the Controller or take commands from it.
- **Low Coupling Principle** - Here is where the design seems less successful, since the Controller is coupled to nearly every class on the page. However, since it is the expert doer of all of these actions, there is a trade-off to be made between coupling, cohesion and expert doer. This diagram represents the best balance between those trade-offs.

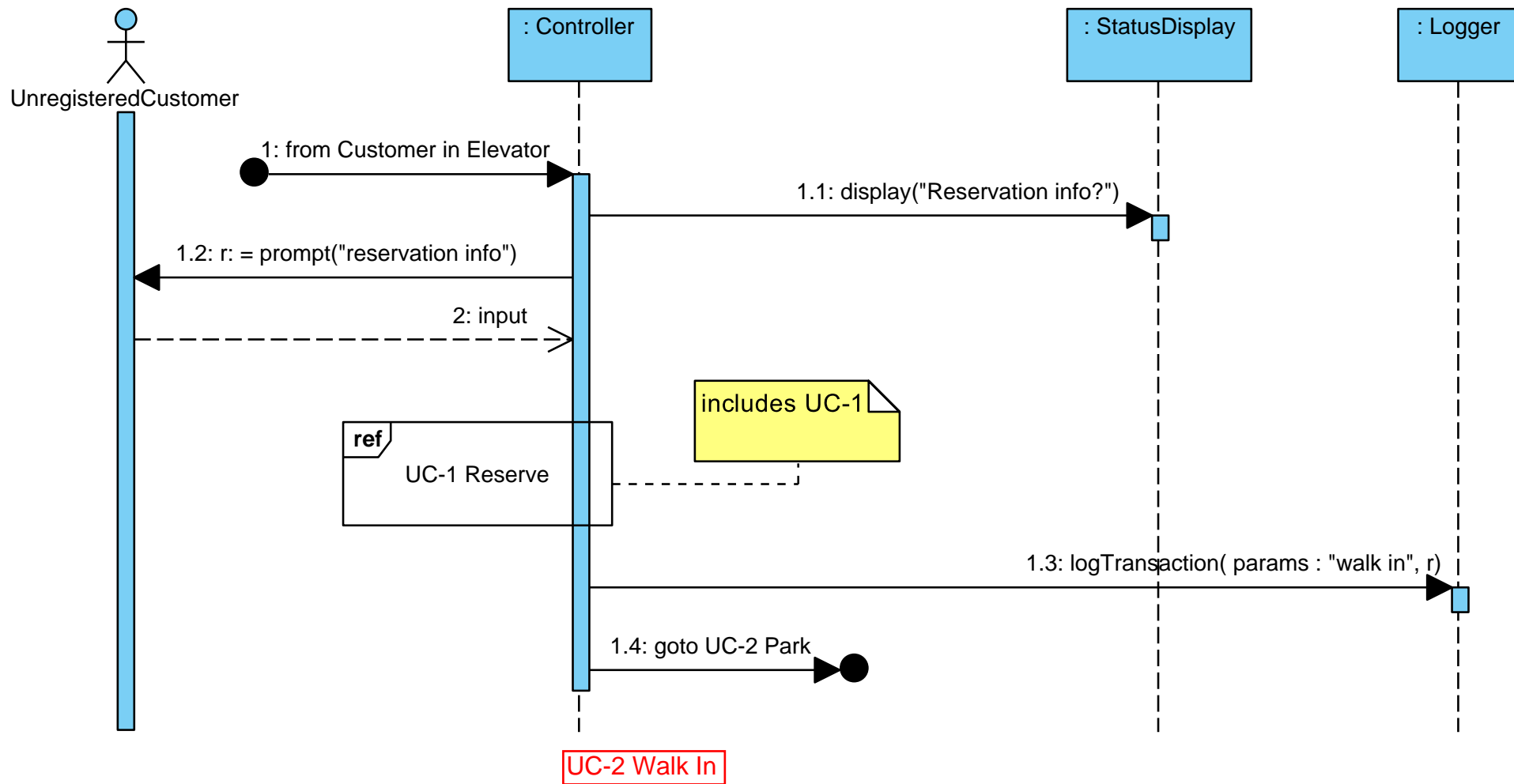
The one thing in this diagram that doesn't quite mesh with the remainder of the system, though, is the Status Display object. Of course, the physical parking garage has a LCD display that shows information to the Customer in the elevator. However, it would be best to simply implement the display as a web page, that way we could standardize how we display information (both on the website when a Customer accesses it from home, and in the parking garage when a Customer comes in to park). This would allow us to eliminate the Status Display class and simply have an Interface Page class with Page Maker. It would make all of our system interaction diagrams more uniform and condensed.

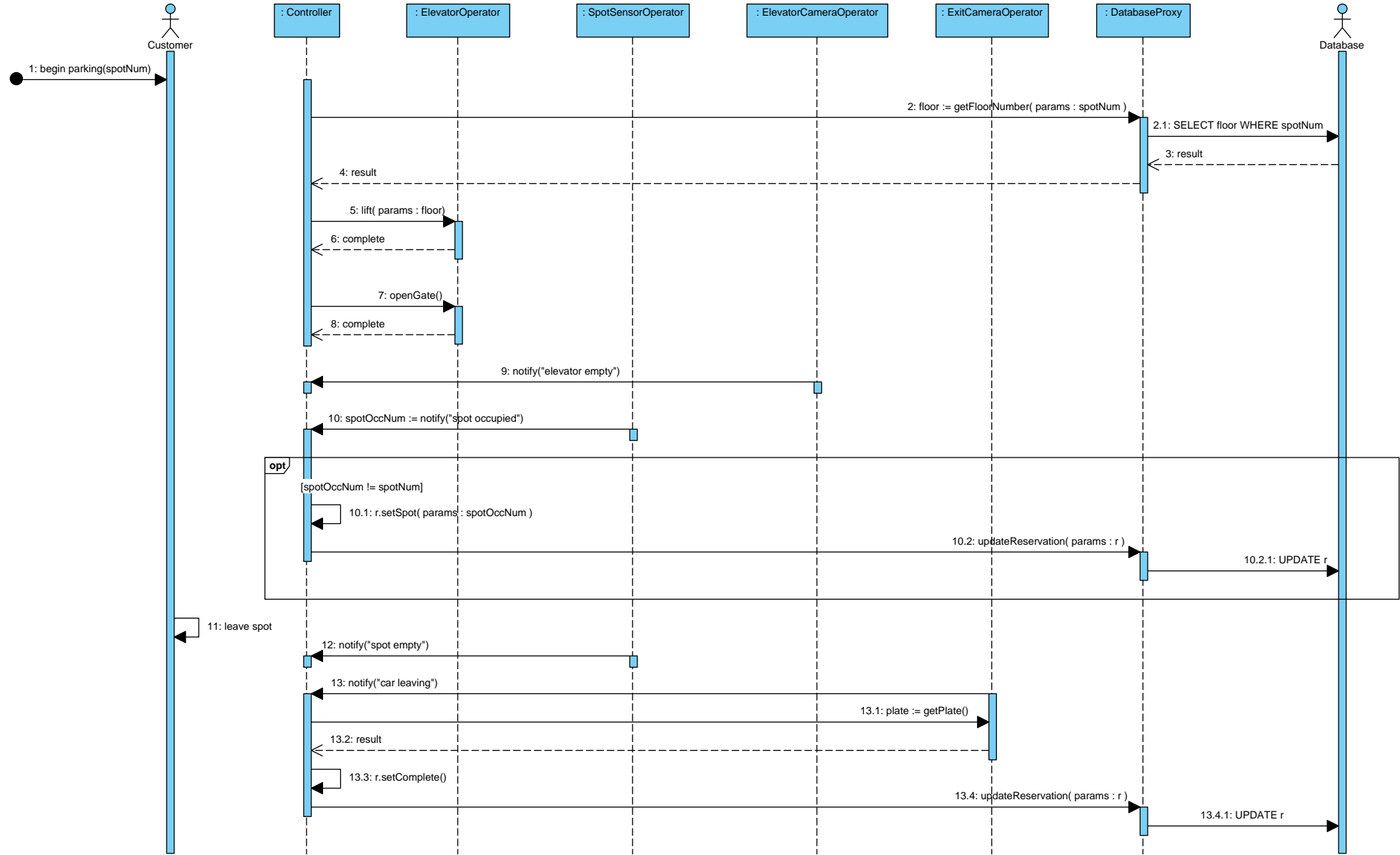


UC-2 Customer in Elevator



UC-2 Registered Customer





UC-2 Park

## UC-3: Manage Account

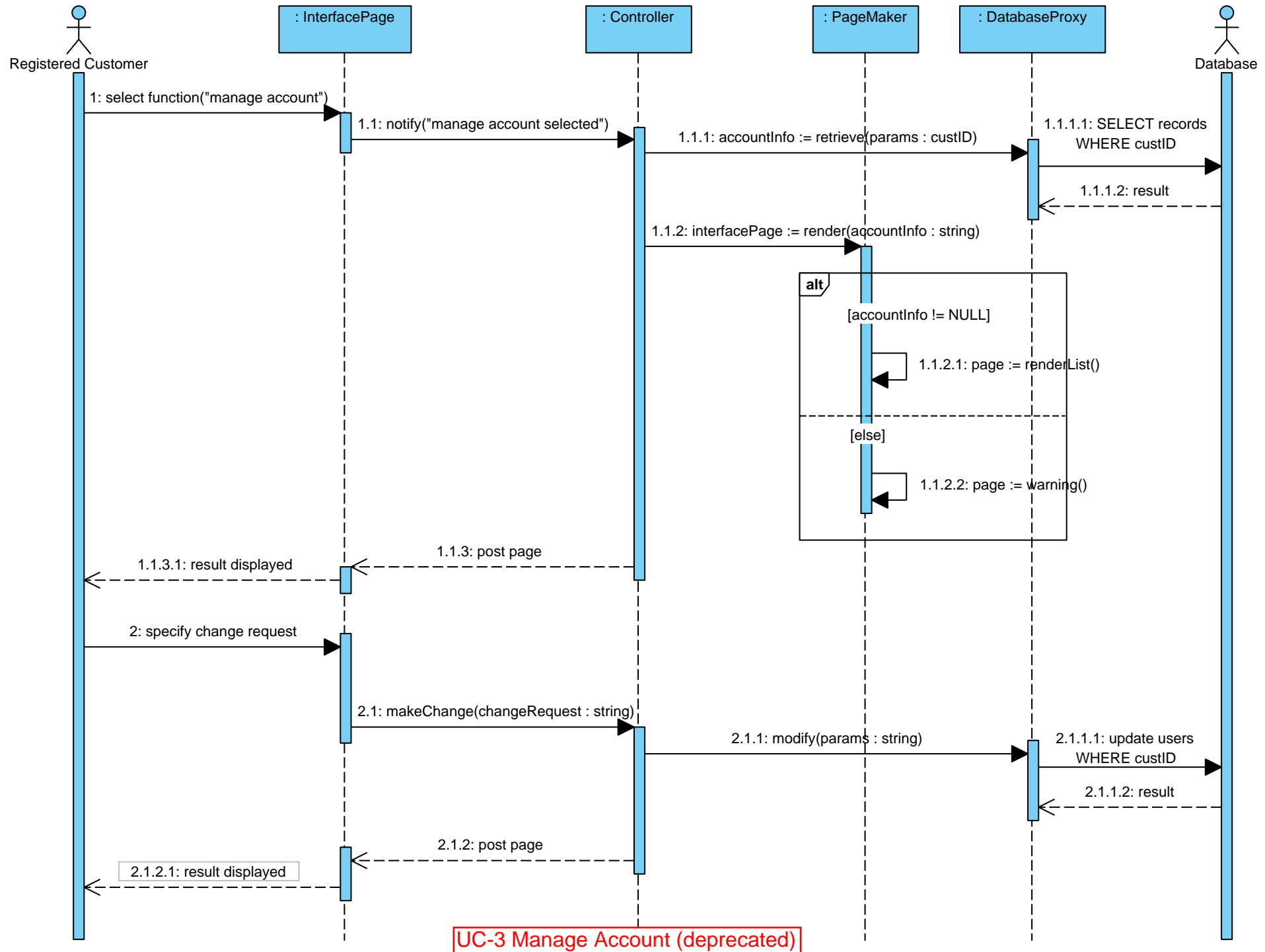
**Goals:** To change account details for a registered customer.

**Process:** The registered customer is prompted to make changes to their current account information. The controller passes the information to the page maker, which updates the account in the database and then displays a page signifying the successful update.

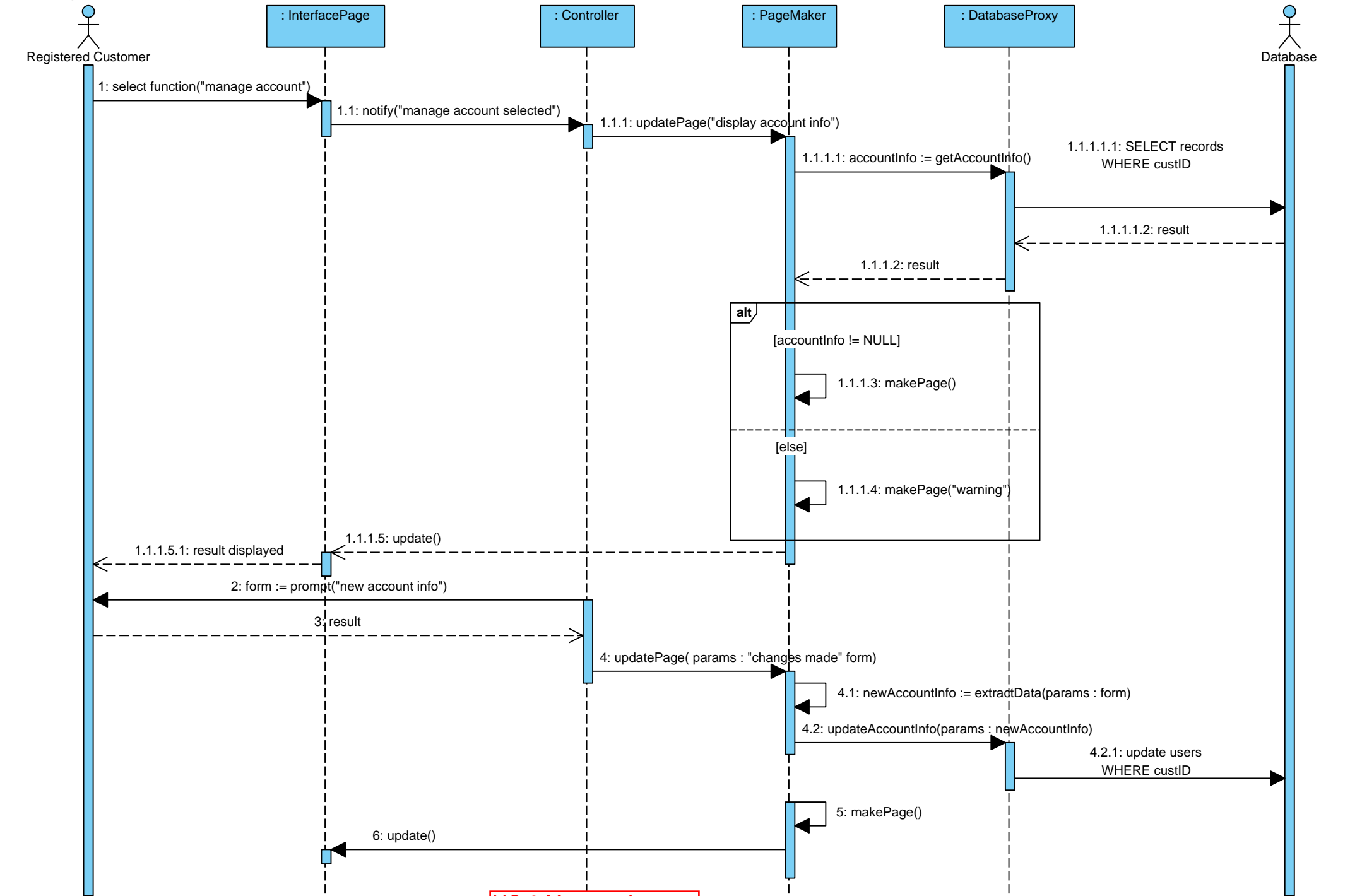
Reasons for diagram selection:

- **Expert Doer Principle** - The controller is the first concept to know what function the registered customer wishes to select and also what info the customer wishes to change. We first thought it would be logical to have the task of communicating with the database proxy but then we decided that that would give too much responsibility to the controller. Therefore we are going against the expert doer principle and having the page maker send requests to the database proxy.
- **High Cohesion Principle** - To satisfy high cohesion, it is necessary to have one concept which has the responsibility of formulating queries for the database and another concept responsible for rendering the pages for display. These two tasks are left to the Database Proxy and Page Maker respectively. If both of these tasks were given to the same concept or given to the Controller to do, then that would be too many tasks for one concept. We choose to split the tasks up to give each concept their own specialized tasks to perform.
- **Low Coupling Principle** - Achieving low coupling would almost directly affect the expert doer principle. The Controller is the first one to know about all of the tasks so it is logical to have it send most of the messages. However this puts a lot of responsibility on the Controller. Since we would rather satisfy the low coupling principle rather than the expert doer principle, we take away some communication responsibilities and give it to the page maker. Overall, this is done to not overburden the Controller which will accumulate too many responsibilities when all of the use cases are put together.

We have included two system interaction diagrams for use case 3. The first diagram gives the responsibility of communication with the Database Proxy to the Controller whereas the second diagram gives the responsibility of communicating with the Database Proxy to the Page Maker. We have decided to go with the second diagram because it reduces the burden put on the Controller therefore satisfying the low coupling principle.







UC-3 Manage Account

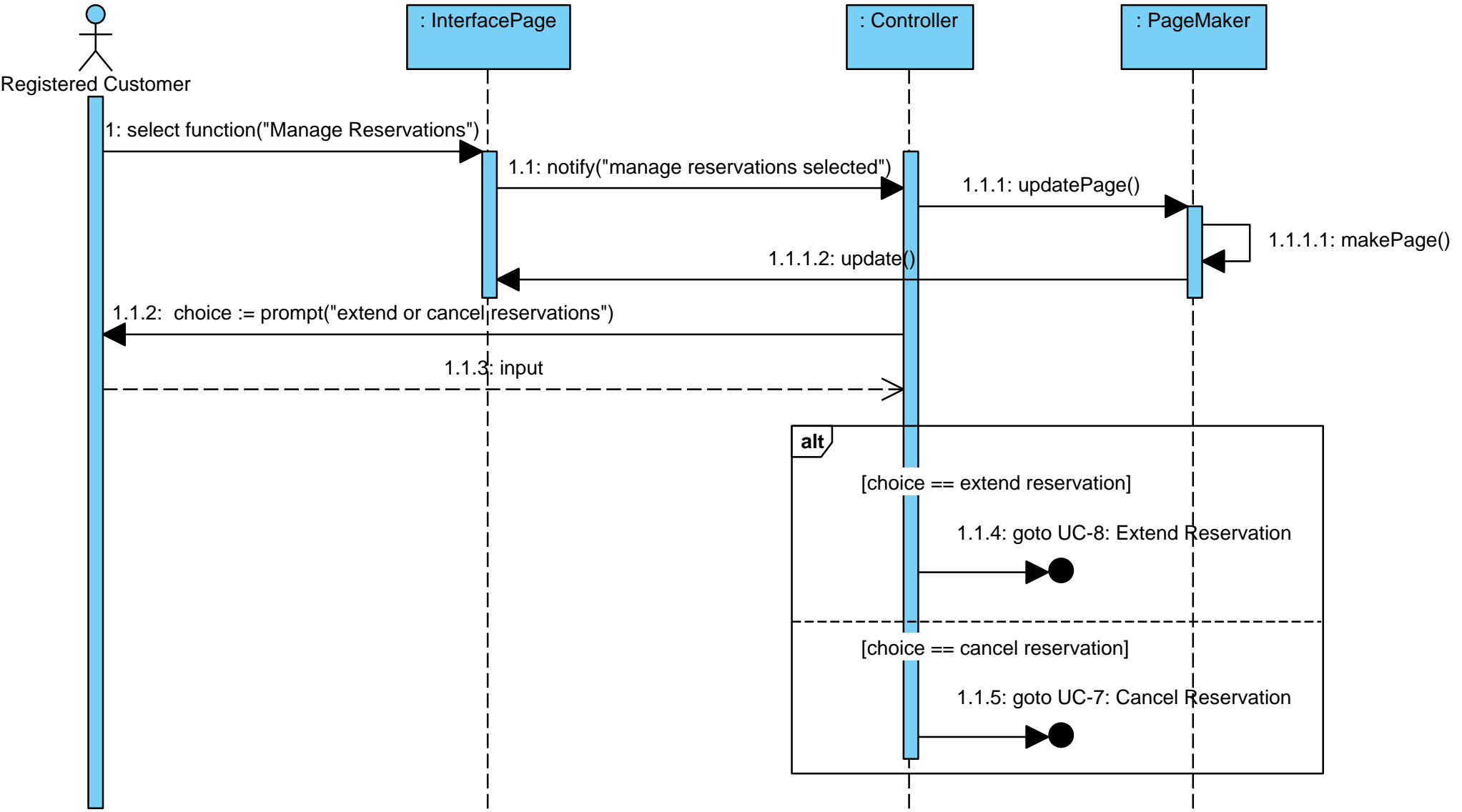
## UC-4: Manage Reservations

**Goals:** To either extend or cancel existing reservations for a registered customer (these are sub-use cases).

**Process:** The registered customer is prompted to select whether to extend or cancel any existing reservations. The Controller receives the choice and then is passes to either system interaction diagram 7 or 8.

Reasons for diagram selection:

- **Expert Doer Principle** - Since this use case only involves two concepts, there is no debate as to who should perform the tasks. The Interface Page is the first to know what selections the user has made, but the Interface Page should remain as a means of displaying information and graphics to the user and nothing else. The Controller is the first worker concept to know about what tasks to be performed and therefor should be the one to perform the tasks.
- **High Cohesion Principle** - This use case is solely a parent use cases for two sub use cases. Therefore there is not much computation to be done. The only computation is the determination of what the user selected and that responsibility is given to the controller since it is the only worker concept in the diagram.
- **Low Coupling Principle** - Once again the Controller should be the one responsible for communicating with other concepts. There is very little messaging to be done in this use case so it is not hard to see why the Controller is given the job.



UC-4 Manage Reservations

## UC-5: Register

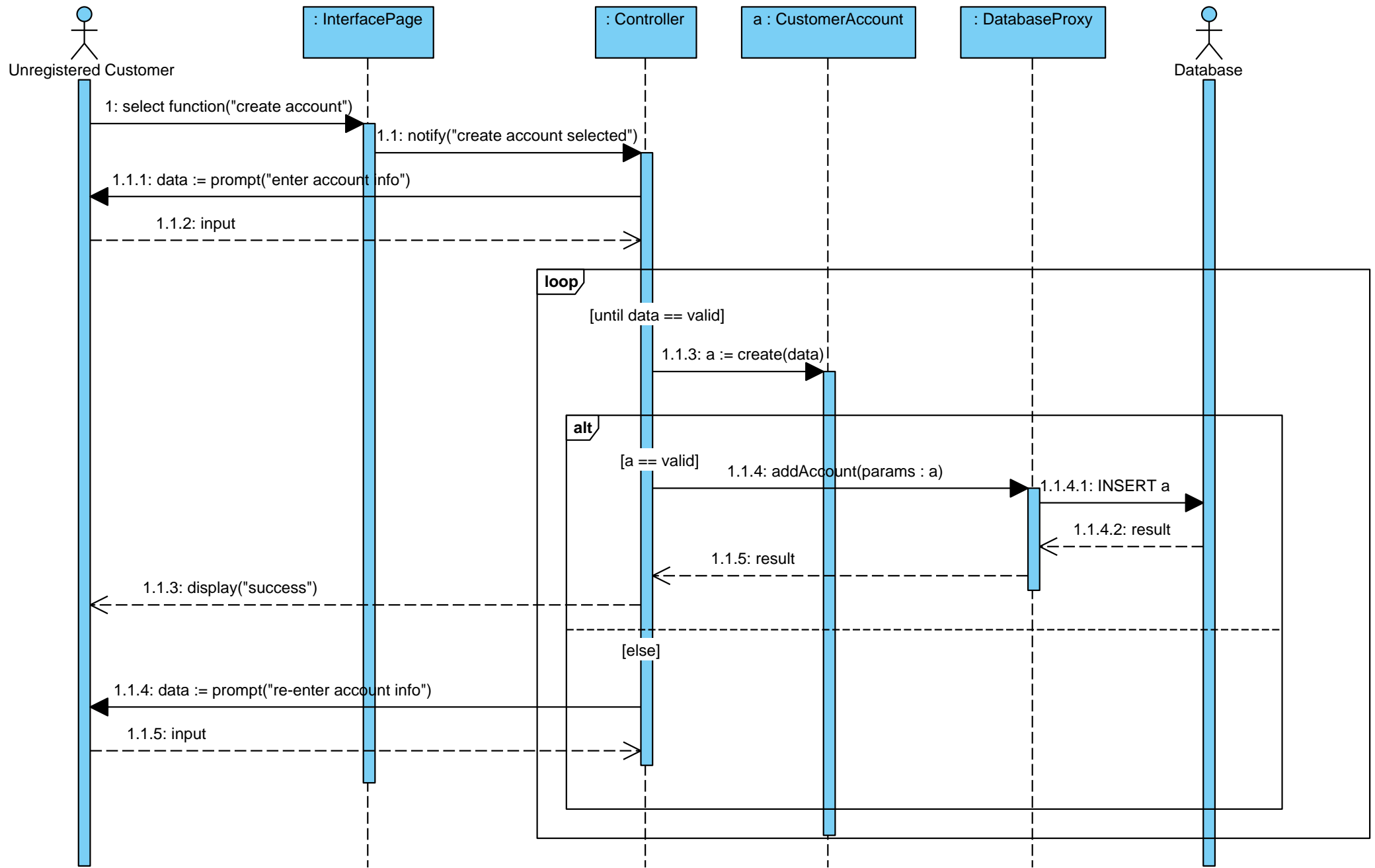
**Goals:** To become a registered customer and be stored in the database.

**Process:** The registered customer is prompted to input their account information. The Controller passes the account information to the Page Maker which creates the account in the database. The Page Maker then displays a page signifying the success.

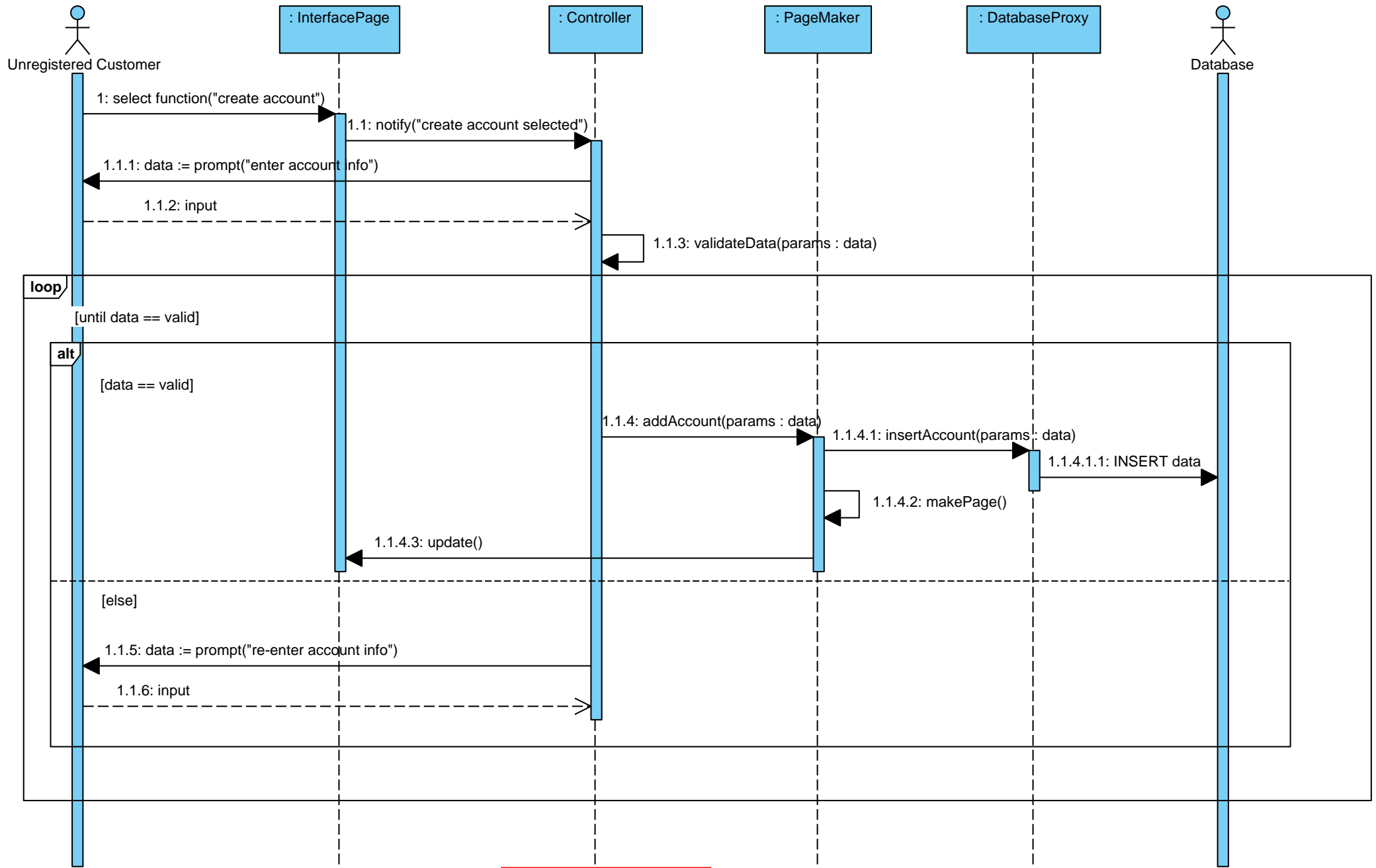
Reasons for diagram selection:

- **Expert Doer Principle** - The Controller is the first worker concept to know what actions the customer wishes to take, however if we were to give it the responsibility of communicating with the Database Proxy, then we would be overburdening it. We decide to have the Page Maker communicate with the database Proxy which is direct violation of the expert doer principle but we deem it is a better option.
- **High Cohesion Principle** - High cohesion is achieved by giving the Database Proxy the responsibility of querying the database. The Database Proxy becomes a very specialized concept associated with a single task and the Controller can then be responsible for communication between the interface page and the Page Maker. The Page Maker is then responsible for making pages and communicating between the Controller and the Database Proxy.
- **Low Coupling Principle** - We have chosen to satisfy the low coupling principle by given some of the Controller's responsibility to the Page Maker. Overall, this keeps the communication strands low for each concept and spreads the work more evenly around.

We have included two system interaction diagrams for use case 5. The first diagram gives the responsibility of communication with the Database Proxy to the Controller whereas the second diagram gives the responsibility of communicating with the Database Proxy to the Page Maker. We have decided to go with the second diagram because it reduces the burden put on the Controller therefore satisfying the low coupling principle.



UC-5 Create Account (deprecated)



UC-5 Create Account

## UC-6: Manage Garage

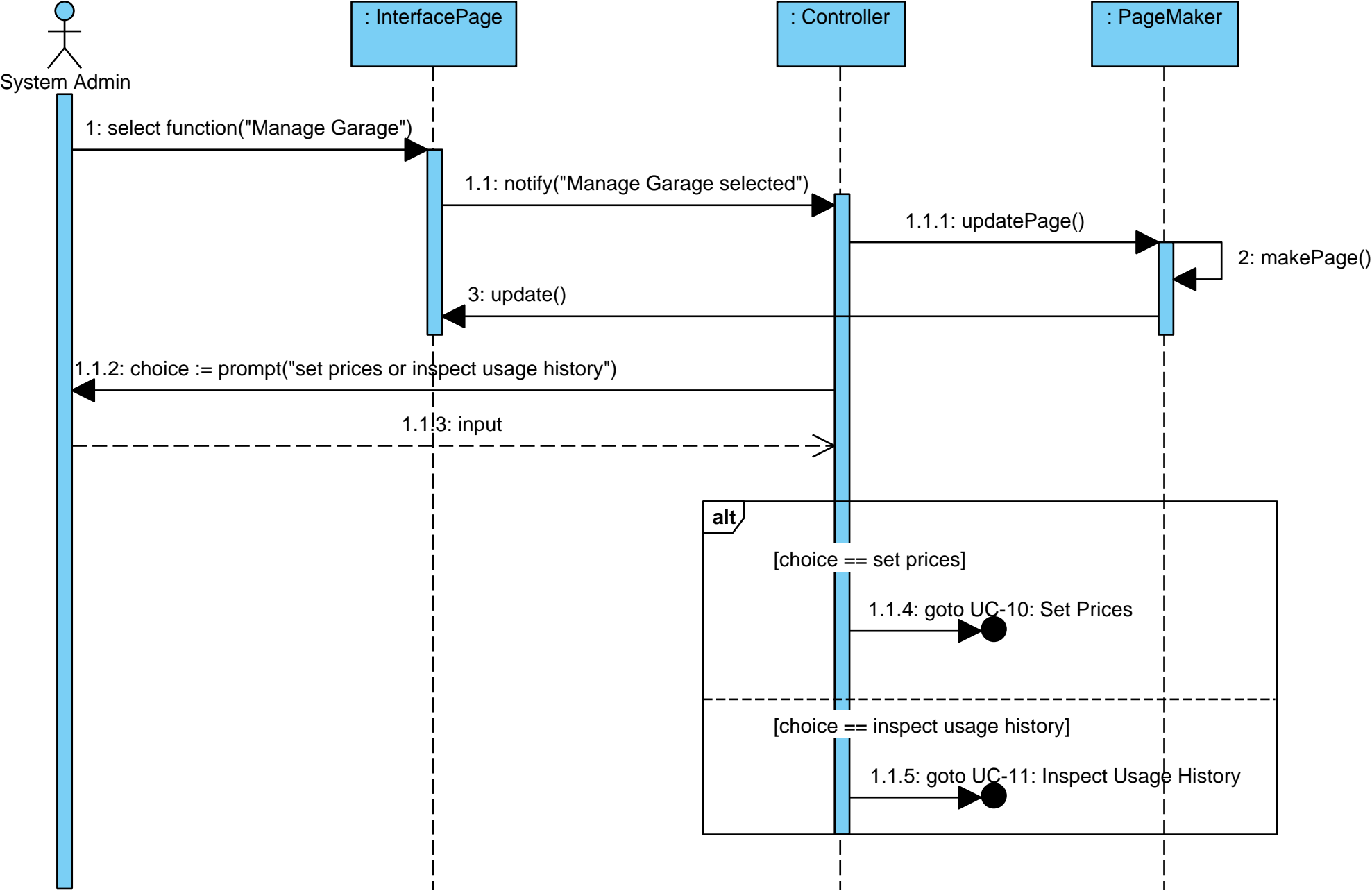
**Goals:** For a system admin to either set prices or view access history of a garage (these are both sub-use cases)..

**Process:** The system admin is prompted to select whether to set prices or view access history for a garage. The Controller receives the choice and then is passes to either system interaction diagram 10 or 11.

Reasons for diagram selection:

- **Expert Doer Principle** - This activity diagram is almost identical to the activity diagram for use case 4 therefore we will employ very similar strategies. This use case only involves two concepts so we simply assign all of the tasks to the Controller which is the only worker concept. The Interface Page is the first to know what selections the user has made, but the Interface Page should remain as a means of displaying information and graphics to the user and nothing else.
- **High Cohesion Principle** - This use case is solely a parent use case for two sub use cases, therefore there is very little computation to be done. The only computation is the determination of what choice the user has made and that responsibility is given to the Controller.

**Low Coupling Principle** - Once again the Controller should be the one responsible for communicating with other concepts. There is very little messaging to be done in this use case so it is not hard to see why the Controller is given the job.



UC-6 Manage Garage



## UC-7: Cancel Reservation

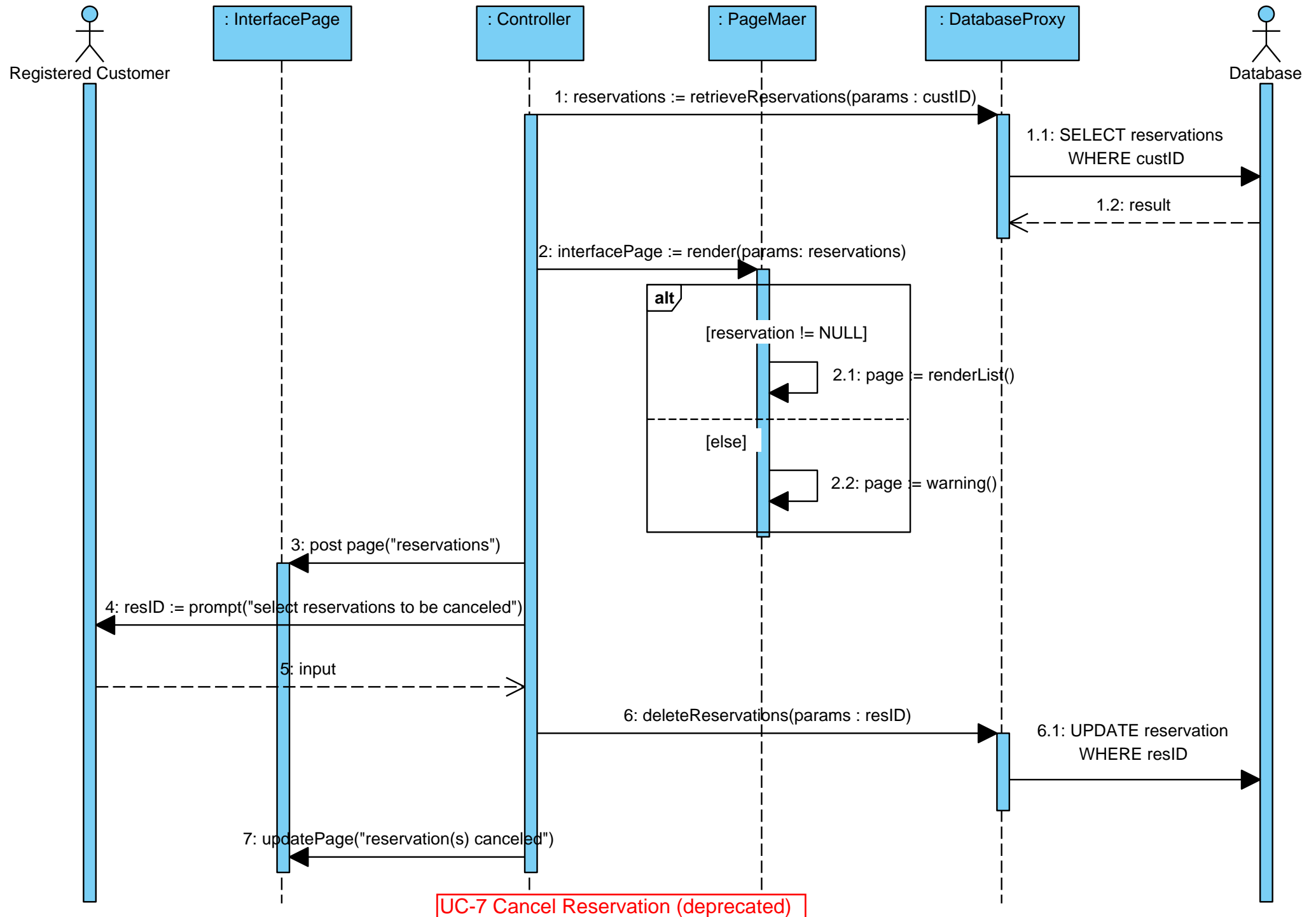
**Goals:** To cancel an existing reservation for a registered customer.

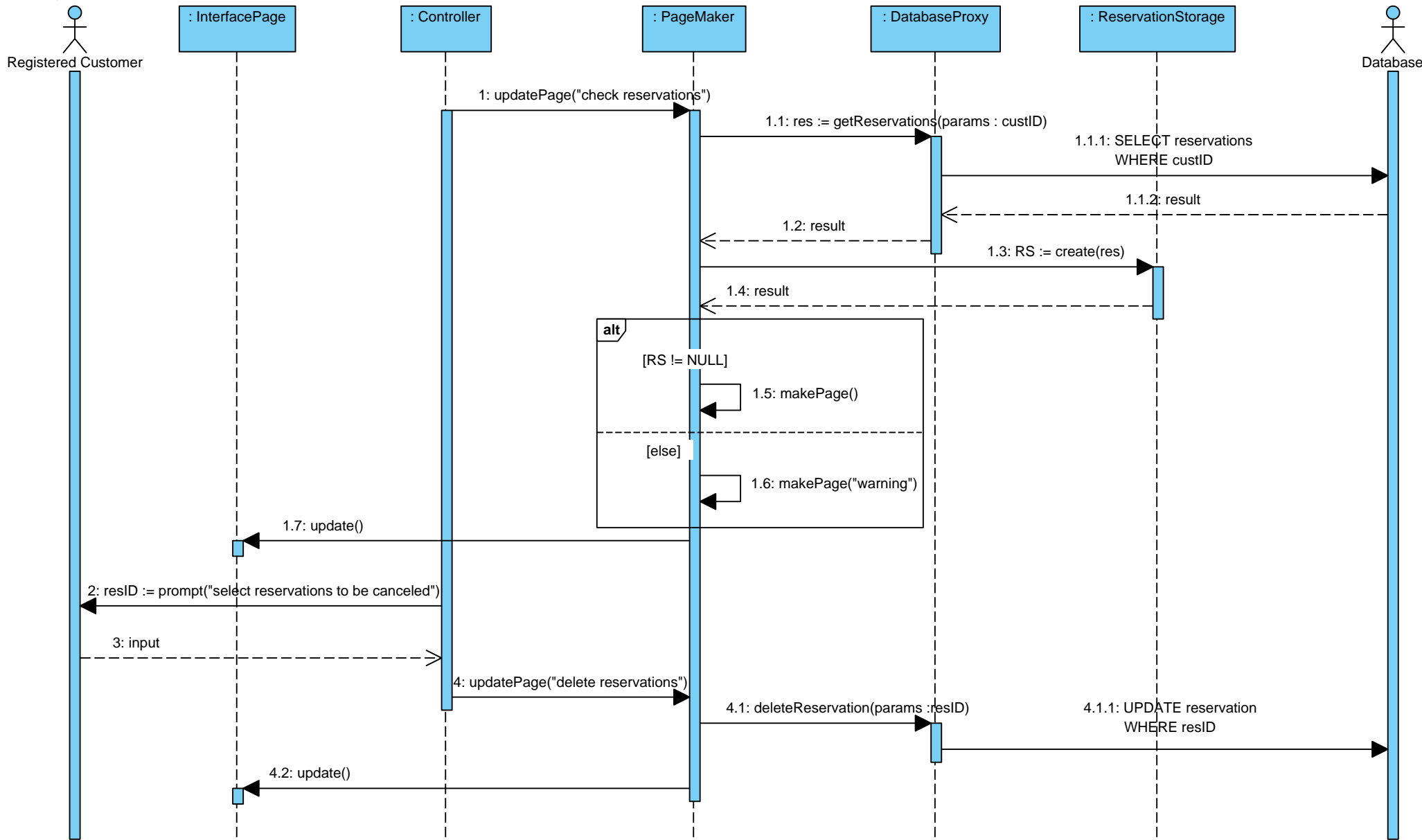
**Process:** The Page Maker requests all existing reservations from the database and displays them for the registered customer. The registered customer then selects which reservations they want to cancel. The Controller then passes this information to the Page Maker which updates the database to list the reservations as “canceled”. The Page Maker then displays the success page to the customer.

Reasons for diagram selection:

- **Expert Doer Principle** - The Controller is the first one to know what tasks to perform, but once again we do not want to let it communicate directly with the Database Proxy. We instead want the Page Maker to have this responsibility. This violates the expert doer principle in exchange for satisfying the low coupling principle.
- **High Cohesion Principle** - To satisfy high cohesion we try not to give too many tasks to the Controller. We do this by giving the responsibility of rendering the pages to the page maker and the responsibility of querying the database to the Database Proxy. By giving those two concepts very specialized tasks, we do not overburden the Controller and yet still keep the other concepts focused on one primary task.
- **Low Coupling Principle** - For this use case we deemed the low coupling principle more vital than the expert doer principle. The Controller’s job is primarily to communicate between the Interface Page and the Page Maker and the Page Maker then communicates with the Database Proxy. This splits the amount of communication evenly between the different worker concepts.

We have included two system interaction diagrams for use case 7. The first diagram gives the responsibility of communication with the Database Proxy to the Controller whereas the second diagram gives the responsibility of communicating with the Database Proxy to the Page Maker. We have decided to go with the second diagram because it reduces the burden put on the Controller therefore satisfying the low coupling principle.





UC-7 Cancel Reservation

## UC-8: Extend Reservation

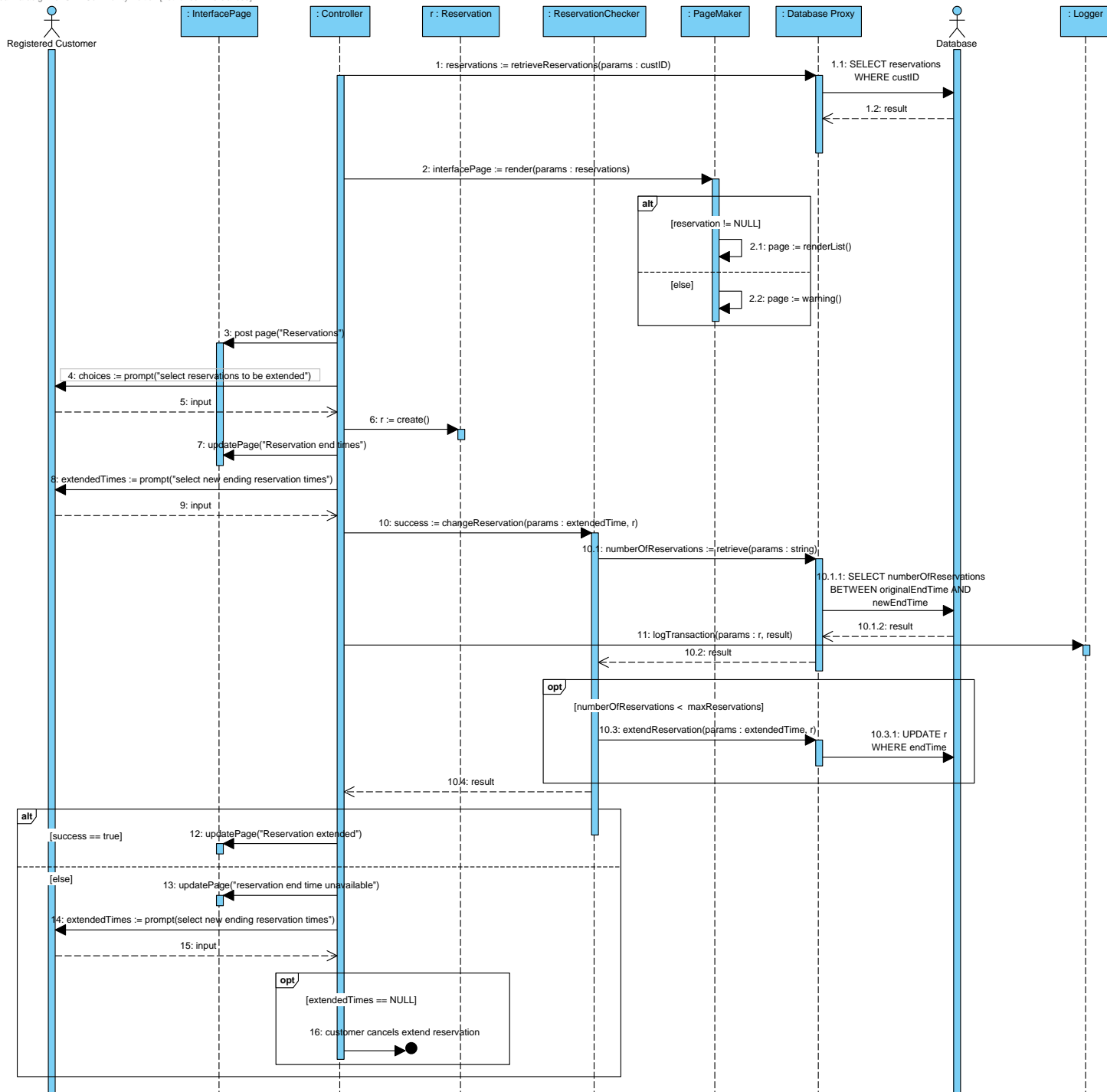
**Goals:** To extend an existing reservation for a registered customer.

**Process:** The Page Maker requests all existing reservations from the database and displays them for the registered customer. The registered customer then selects which reservations they want to extend. The Controller then passes this information to the Page Maker which then passes the information to the Reservation Checker which compares the number of reservations in the extended time to the maximum allowed number of reservations. If there are available reservations, then the Reservation Checker tells the Page Maker if it was a success or a failure and updates the page to display the information.

Reasons for diagram selection:

- **Expert Doer Principle** - This diagram requires some real thought because two worker concepts have a lot of responsibility. The Controller is the first to know what decisions the customer has made, but the Reservation Checker is the first to know from the Database Proxy how many reservations are available at the requested time. For the previous reason, we have give the Reservation Checker the tasks of deciding whether a reservation can be extended and then telling the Database Proxy to extend the reservation in the Database.
- **High Cohesion Principle** - The expert doer principle for this use case helps to ensure high cohesion. By splitting the work to be done between the Controller and Reservation Checker, the Controller is not asked to do every computational task and the Reservation Checker can still be a specialized concept. The alternate choice would be to have the Controller perform all of the logic to determine if a reservation can be extended, but that takes away most of the responsibility of the Reservation Checker and leaves the Controller with too many tasks to perform.
- **Low Coupling Principle** - To ensure low coupling, the communication responsibilities are split between the Controller the Reservation Checker and the Page Maker. The Controller communicates between the customer the interface page, the Reservation Checker communicates between the Database Proxy and the Database and the Page Maker communicates between the Controller and the Reservation Checker. By having both concepts split the messaging work, low coupling is achieved.

We have included two system interaction diagrams for use case 8. The first diagram gives the responsibility of communication with the Reservation Checker to the Controller whereas the second diagram gives the responsibility of communicating with the Reservation Checker to the Page Maker. We have decided to go with the second diagram because it reduces the burden put on the Controller therefore satisfying the low coupling principle.



UC-8 Extend Reservation  
(deprecated)

## UC-8 Extend Reservation

## UC-9: Authenticate User

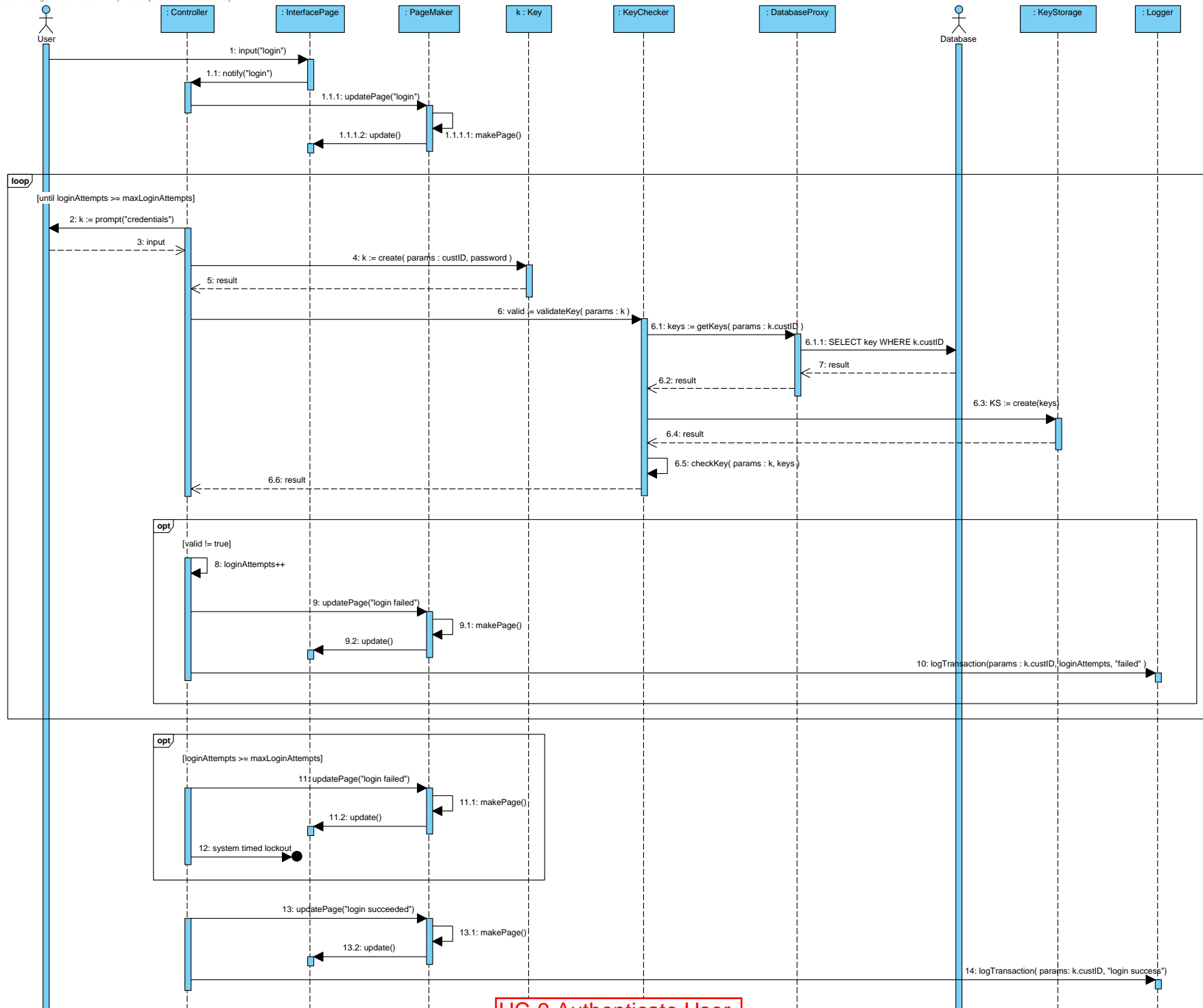
**Goals:** To determine if a user is registered with the system and has an account in the database.

**Process:** The user accesses the Interface Page via a web browser, selects log in, and enters their credentials (email and password). The Controller creates a Key from this information which it passes to Key Checker, which then handles the verification of the user. The user is allowed a maximum number of login attempts, and must either authenticate within those given number of logins or be locked out of the system for a pre-determined period.

Note that for this diagram, User is collectively Registered Customer and System Admin, since both of those types of user can authenticate themselves with the system.

Reasons for diagram selection:

- **Expert Doer Principle** - We satisfy this principle by dividing up the work between the Controller and the Key Checker. Key Checker handles verifying the user input key (combo of customer ID and password), dealing with database queries, and checking the input key against those in the database. Controller, which is connected to all of the boundary components of the system, such as the display and user inputs. The Controller coordinates the user input with the Key Checker and the user.
- **High Cohesion Principle** - Each class in our system interaction diagram has a large portion of the responsibility, mainly split between the Controller and the Key Checker. The Controller coordinates between the user input and the Key Checker, and the Key Checker handles all necessary verification.
- **Low Coupling Principle** - Again, the expert doer principle here helps ensure that we have low coupling, since the object that knows is performing the action. This cuts down on the coupling between classes, since Controller merely ferries data between the Key Checker and the user, and the Key Checker handles all dealings with the Database. This keeps our system very de-coupled.



UC-9 Authenticate User



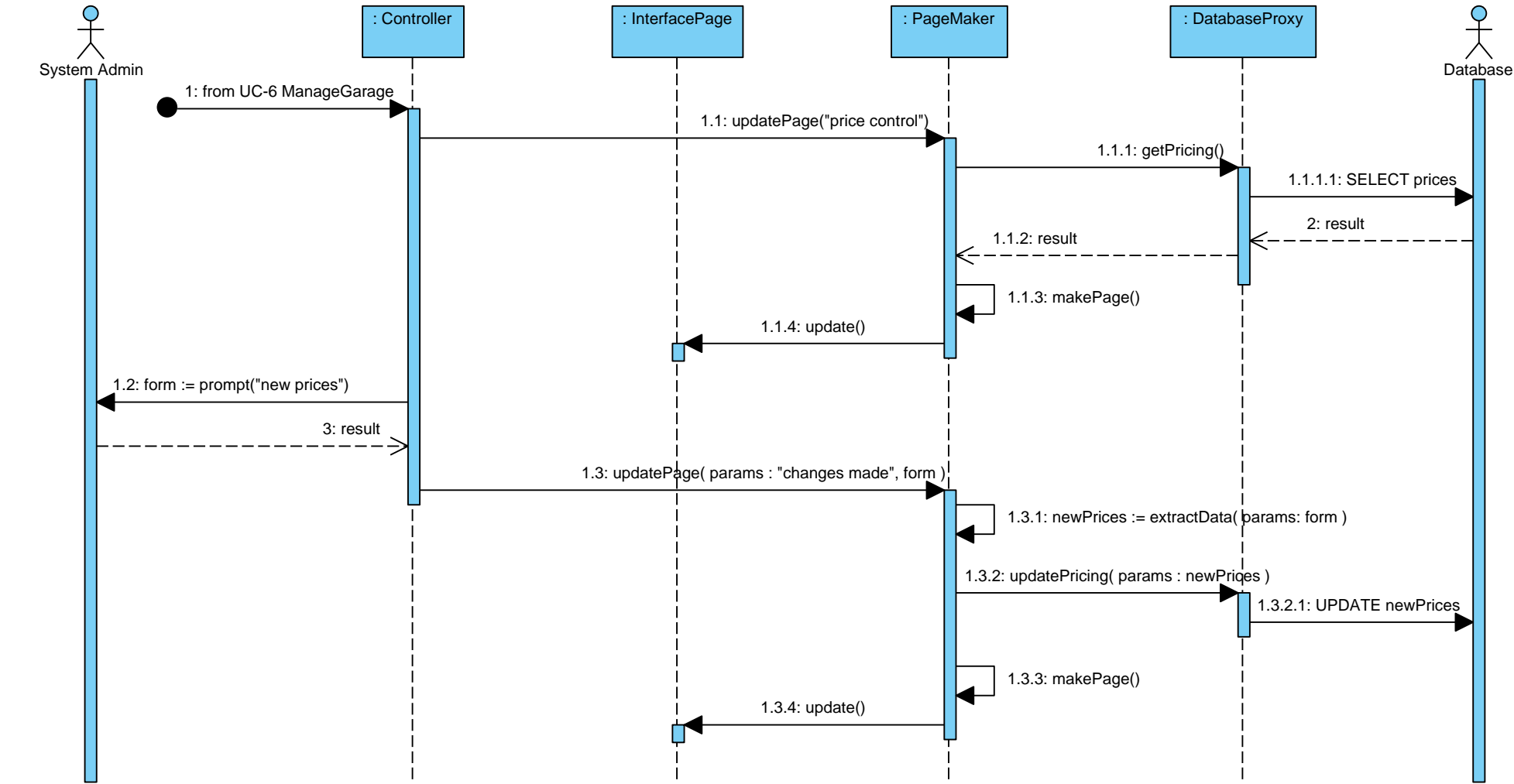
## UC-10: Set Prices

**Goal:** To update the prices for parking at the garage.

**Process:** The system admin requests via a web browser to change the prices for a garage, and is given a web page containing the old pricing information and a form to change to newer prices. Controller takes this form and passes it to the Page Maker, which extracts the new info from it and updates the Database with that pricing data. The web page the system admin sees is then updated with the new prices and a confirmation of the changes.

Reasons for diagram selection:

- **Expert Doer Principle** - The expert doer principle is satisfied well here, because only the objects that need to know information are performing work on it. The Page Maker interacts directly with the Database to retrieve information it needs to create HTML pages to present via the Interface Page. The Controller deals with the orchestration of tasks between the System Admin and the Page Maker. Therefore, the one who knows is the one performing the action.
- **High Cohesion Principle** - In this system interaction diagram there is very high cohesion, since Controller and Page Maker each have varied tasks, and handle most of the work in the specific diagram. Page Maker has a high level of cohesion because it interacts directly with the Database Proxy, giving it both the responsibility to create the Interface Page and also retrieve the information included in it.
- **Low Coupling Principle** - Although the coupling in this diagram is not fantastic, there are lower levels than if we had relied upon the Controller to access the Database through Database Proxy and pass that information on to the Page Maker. Allowing no link between Controller and Database Proxy greatly reduces the coupling of this system.



UC-10 Set Prices

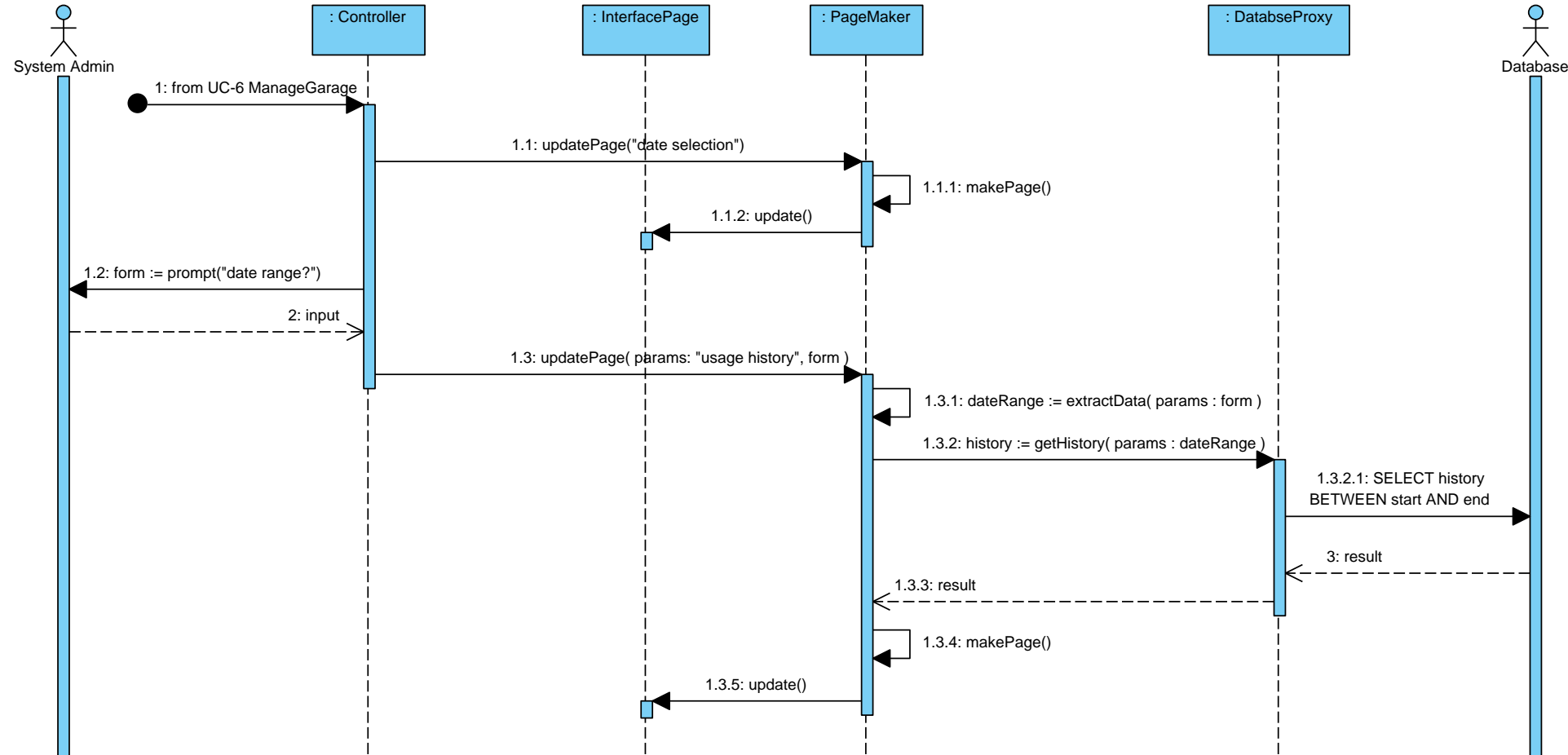
## UC-11: Inspect Usage History

**Goal:** To view the usage history of the parking garage.

**Process:** The system admin requests to view the parking garage usage history, and the Controller requests from Page Maker that the history be pulled from the Database for a certain range of dates input by the system admin. The Page Maker retrieves this data and creates an HTML page to display it to the system admin for review.

Reasons for diagram selection:

- **Expert Doer Principle** - As with UC-10 Set Prices, this use case is very similar, except that it only allows the Admin to view information about the garage, not edit it. Therefore, the structure is very similar, with the same expert doers handling their tasks (Controller and Page Maker).
- **High Cohesion Principle** - As with UC-10, there is a very high level of cohesion here since we have assigned so many responsibilities to Page Maker, and Controller as well.
- **Low Coupling Principle** - The coupling here is the same as it was in UC-10, with the Page Maker linking to both the Controller and the Database Proxy, but removing any need for the Controller to talk directly with the Database Proxy.



UC-11 Inspect Usage History

# Class Diagram and Interface Specification

## Class Diagrams

For this project, we plan to take advantage of a PHP framework known as Kohana<sup>[4]</sup>. Therefore, our class diagrams will be a composition between classes that come directly from tables in the Database (see next section, Data Types and Operation Signatures). In addition, we develop a class diagram from our system interaction diagrams. The marriage of these two will be the basis for our system.

The advantage to using the Kohana framework in this way, we can now run methods against objects in our program rather than SQL queries against the database. From a programming perspective, this is ideal because it allows to always keep an object-oriented view about our program (treating tables in the database as objects in the program). However, it makes our system interaction diagrams slightly more difficult to translate into a class diagram.

Therefore, we define both a class diagram from our system interaction diagram and also a class diagram for our Kohana framework mapped tables.

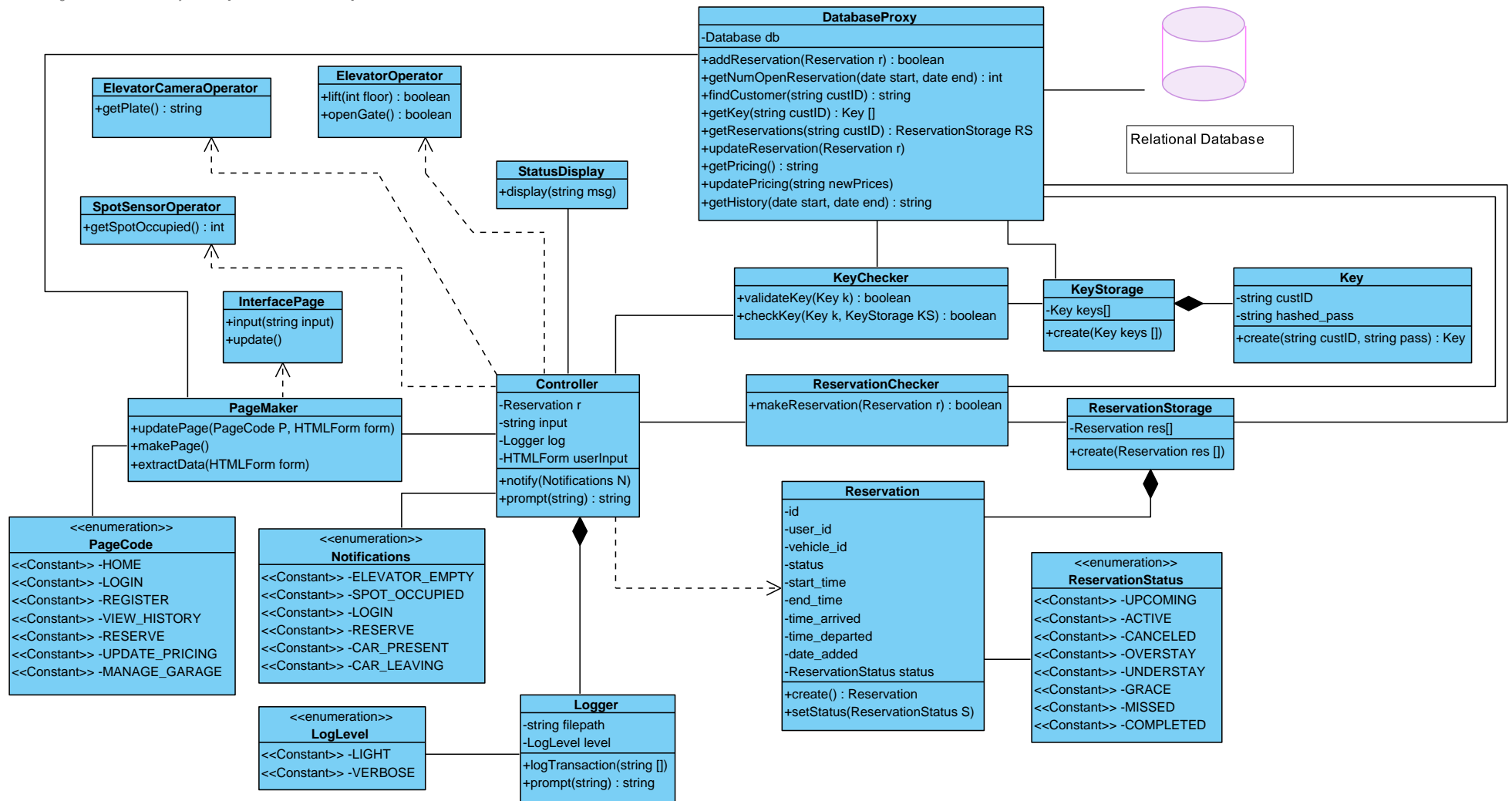
To alleviate a lot of the coding needed to implement our system, we will be using an existing, open source PHP framework, Kohana version 3.1. The Kohana framework will allow for the MVC (model, view, controller) architecture. Our business logic exists inside our models. Within Kohana, there is an ORM (object relational mapping) library that abstracts a lot of the database queries as objects. Within the ORM already exist a lot of common methods, such as save(), create(), edit(), etc. We can talk about the class diagrams later on during the meeting. The way I see it, every class should be responsible for creating itself. This is a possible implementation, but again, there are many ways to do this.

```
class Model_Reservation extends ORM
{
    protected $_belongs_to = array(
        'user' => array('model' => 'user'),
    );

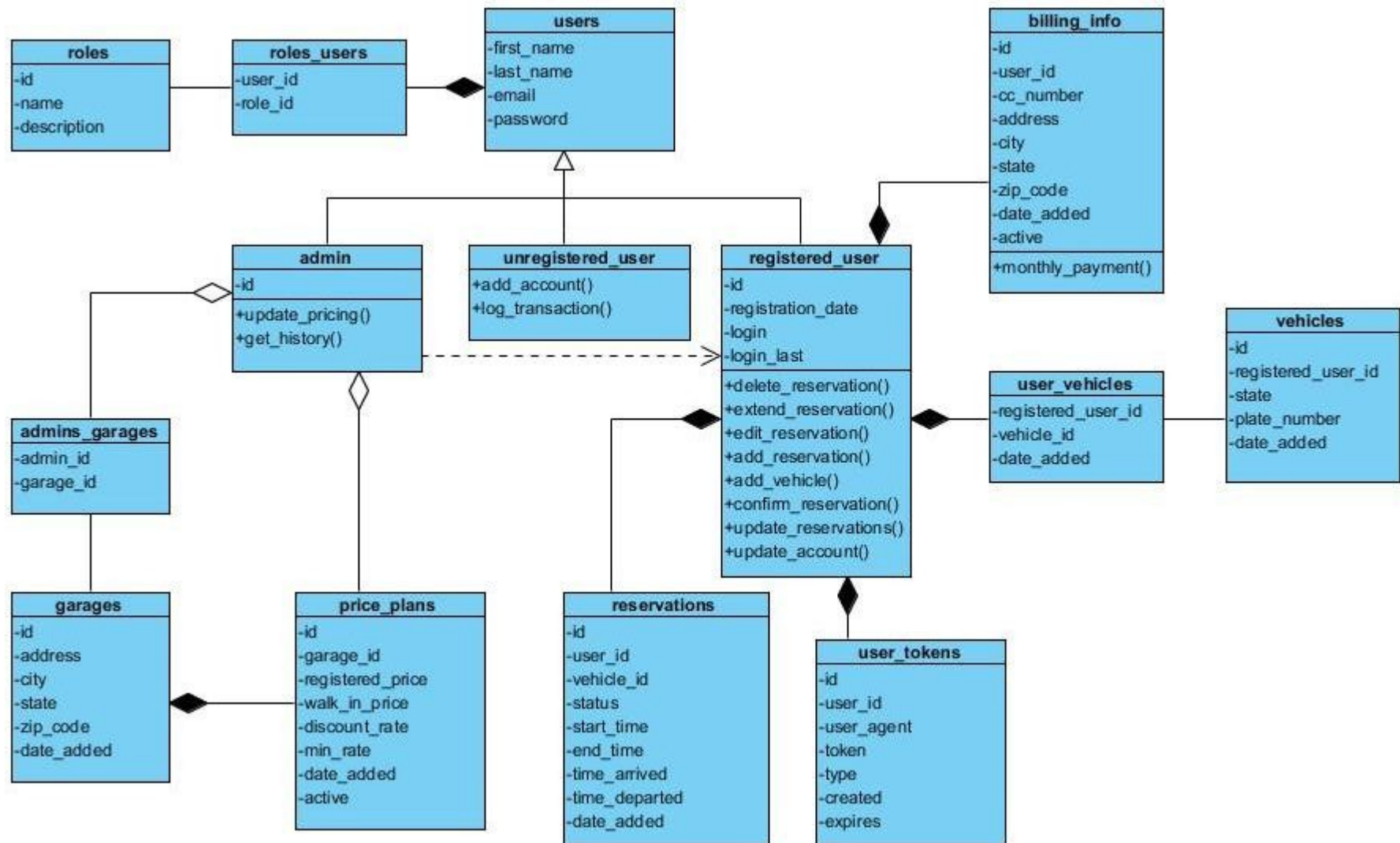
    public function create_reservation(array $values)
    {
        // Will throw an exception if validation fails
        $this->values($values)->create();

        return TRUE;
    }
}
```

```
}  
  
class Model_User extends ORM  
{  
    protected $_has_many = array(  
        'reservations' => array('model' => 'reservation'),  
    );  
  
    public function add_reservation(array $values)  
    {  
        $reservation = new Model_Reservation;  
        $reservation->user_id = $this->id;  
  
        return $reservation->create_reservation($values);  
    }  
}
```



Class Diagram from System Interaction Diagrams



Kohana Framework Class Diagram



## Data Types and Operation Signatures

The Kohana framework allows us to dynamically map tables from our database directly to object types in our system, which we define below.<sup>[4]</sup>

**admins:** (users with admin rights)

Attributes:

admins	
id:	INTEGER
first_name:	VARCHAR(32)
last_name:	VARCHAR(64)
email:	VARCHAR(127)
password_1:	VARCHAR(64)

Operations:

*update\_pricing(params : newPrices );*

This method allows only admins to adjust the pricing in the price\_plans class.

*get\_history(params : dataRange );*

This method allows the admins to check the usage history of users and corresponding garages.

**admins\_garages:** (join table for admins and the garages they use)

Attributes:

admins_garages	
admins_id:	INTEGER
garages_id:	INTEGER

Operations:

There are currently no operations for this class.

**billing\_info:** (Billing information for each customer)

Attributes:

billing_info	
id: INTEGER(10)	
registered_user_id: INTEGER (FK)	
cc_number: BIGINT(16)	
address: VARCHAR(80)	
city: VARCHAR(60)	
state: CHAR(2)	
zip_code: MEDIUMINT(5)	
date_added: DATE	
active: TINYINT(1)	
billing_info_FKIndex1	
registered_user_id	

Clarifications: *cc\_number*: is the registered users credit card number

*active*: represents whether or not this card will be billed at the end of the billing cycle.

Operations:

*monthly\_payment( params: active, cc\_number);*

The monthly payment operation automatically deducts the balance of the linked registered user.

**garages:** (garages that can be parked at)

Attributes:

garages	
id: INTEGER	
address: VARCHAR(80)	
city: VARCHAR(60)	
state: CHAR(2)	
zip_code: MEDIUMINT(8)	
date_added: DATE	

Operations:

There are currently no operations for this class.

**price\_plans:** (pricing for each individual garage)

Attributes:

price_plans	
id: INTEGER	
garages_id: INTEGER (FK)	
registered_price: FLOAT	
walk_in_price: FLOAT	
discount_rate: FLOAT	
min_price: FLOAT	
date_added: DATE	
active: TINYINT(1)	
price_plans_FKIndex1	
garages_id	

Clarifications: *registered\_price*: price registered customers pay.

*walk\_in\_price*: price walk in costumers pay.

*discount\_rate*: the rate at which registered customers earn discounts.

*min\_price*: the minimum allowable price to charge any customer.

*active*: Only one price can by active at a moment.

Operations:

There are currently no operations for this class.

**registered\_users:** (users that have registered and can now make reservations)

Attributes:

registered_user	
id: INTEGER	
first_name: VARCHAR(32)	
last_name: VARCHAR(64)	
email: VARCHAR(127)	
password_1: VARCHAR(64)	
registration_date: DATETIME	
logins: INTEGER	
last_login: DATETIME	

Operations:

*add\_reservation( params: r );*

The user will use this method to add a reservation to park at a garage.

*delete\_reservation(params: resID);*

The user will use this method to remove a reservation that they have already signed up for.

*edit\_reservation( params: newTime, resID );*

The user will use this method to change the time of a reservation they have already signed up for.

*extend\_reservation( params: extendTime, r);*

The user will use this method to add time to a current reservation.

*confirm\_reservation(params: resID );*

The user will use this method to confirm a reservation that they have made.

*update\_account\_Info( params: newAccountInfo);*

The user will use this method to update their accounts information.

*update\_reservations( params: r);*

When the user parks in a garage this method is used to update the database.

*add\_vehicle(params: newVehicleInfo);*

The user will use this method to register their vehicle.

**reservations:** (reservations that are in the system)

Attributes:

reservations	
🔑	id: INTEGER
🔑	registered_user_id: INTEGER (FK)
🔑	garage_id: INTEGER
🔑	vehicle_id: INTEGER
🔹	status_reserv: ENUM
🔹	start_time: DATETIME
🔹	end_time: DATETIME
🔹	time_arrived: DATETIME
🔹	time_departed: DATETIME
🔹	date_added: DATE
🔹	last_edited: DATE
📁	reservations_FKIndex1
🔑	registered_user_id

Clarifications: *garage\_id*: the garage where the user will park.

*vehicle\_id*: the vehicle the user is requesting parking for.

*status*: the current status of the reservation.

*start\_time*: scheduled start time.

*end\_time*: scheduled end time.

*time\_arrived*: actual start time.



*time\_departed*: actual end time.

Operations:

There are currently no operations for this class.

**roles:** (roles of the different users in our system)

Attributes:



roles	
	id: INTEGER
	name: VARCHAR(32)
	description: VARCHAR(255)

Operations:

There are currently no operations for this class.

**roles\_users:** (a join class between roles and the users)

Attributes:





roles_users	
	role_id: INTEGER
	user_id: INTEGER

Operations:

There are currently no operations for this class.

**unregistered\_users:** (users that have not registered in our system but have the ability too)

Attributes:

unregisterd_user	
	first_name: VARCHAR(32)
	last_name: VARCHAR(64)
	email: VARCHAR(127)
	password_1: VARCHAR(64)

Operations:

*add\_account( params: data);*

This method allows an unregistered user to sign up to become a registered user, enabling them to make reservations.

*log\_transaction( params: "walk in",r);*

This method is used when unregistered customers walk in and the systems database needs to be updated.

**user\_tokens:** (class for collecting data when users log into the system)

Attributes:

user_tokens	
id: INTEGER	
registered_user_id: INTEGER (FK)	
user_agent: VARCHAR(40)	
token: VARCHAR(40)	
type_token: VARCHAR(100)	
created: INTEGER	
expires: INTEGER	
<i>user_tokens_FKIndex1</i>	
registered_user_id	

Operations:

There are currently no operations for this class.

**user\_vehicles:** (a join class between users and their vehicles)

Attributes:

user_vehicles	
registered_user_id: INTEGER	
vehicle_id: INTEGER	
date_added: DATE	

Operations:

There are currently no operations for this class.

**vehicles:** (vehicles that users have registered and can use for reservations)

Attributes:

vehicles	
id: INTEGER	
registered_user_id: INTEGER (FK)	
plate_number: VARCHAR(10)	
state: CHAR(2)	
date_added: DATE	
<i>vehicles_FKIndex1</i>	
registered_user_id	

Clarifications: *registered\_user\_id*: the user who added the vehicle.

Operations:

There are currently no operations for this class.

# System Architecture and System Design

## Architectural Styles

The structure used in our project resembles the Event-Driven architecture. The EDA is a pattern that promotes the production, detection, consumption of, and reaction to certain events.

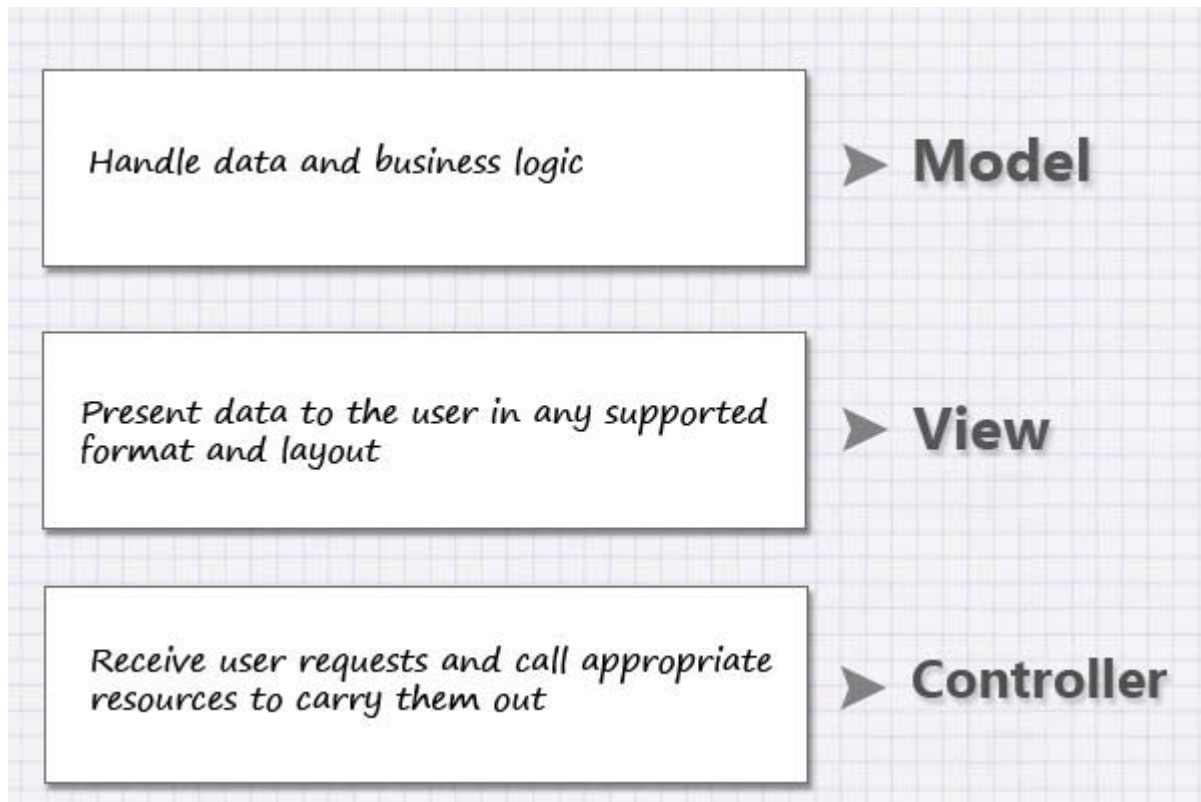
Our software is based on the customers requesting to park their vehicles in our garage.

Therefore our system architecture is based on the event of the customer wanting to reserve a parking space.<sup>[1]</sup>

Different entities change state in our system. A reservation can have multiple states during its life cycle. When a customer cancels a reservation, its state changes from “active” to “canceled”. If a customer fails to leave the garage after the reservation time has ended, its state changes from “active” to “overstay”. The following table sums up the events and states for a reservation.

Events for Reservation	Pre-State	Post-State
Created	None	“upcoming”
Canceled	“upcoming”	“canceled”
Extended	“active”, “upcoming”	Same as Pre-State
Overstay	“active”	“overstay”
Understay	“active”	“understay”
Grace	“upcoming”	“grace”
Missed	“grace”	“missed”
Completed	“active”, “overstay”, “understay”, “missed”	“completed”

The framework we will be implementing in “Park-A-Lot” is a HMVC ( Hierarchical Model View Controller) framework. The main point in MVC is straight forward: the following responsibilities must be clearly separated.



The controller deals with the user requests. It controls and coordinates the things needed in order to execute the user request. The model consists of the data and the rules or policies regarding the data. The view creates a way to represent the data obtained from the model.<sup>[2]</sup>



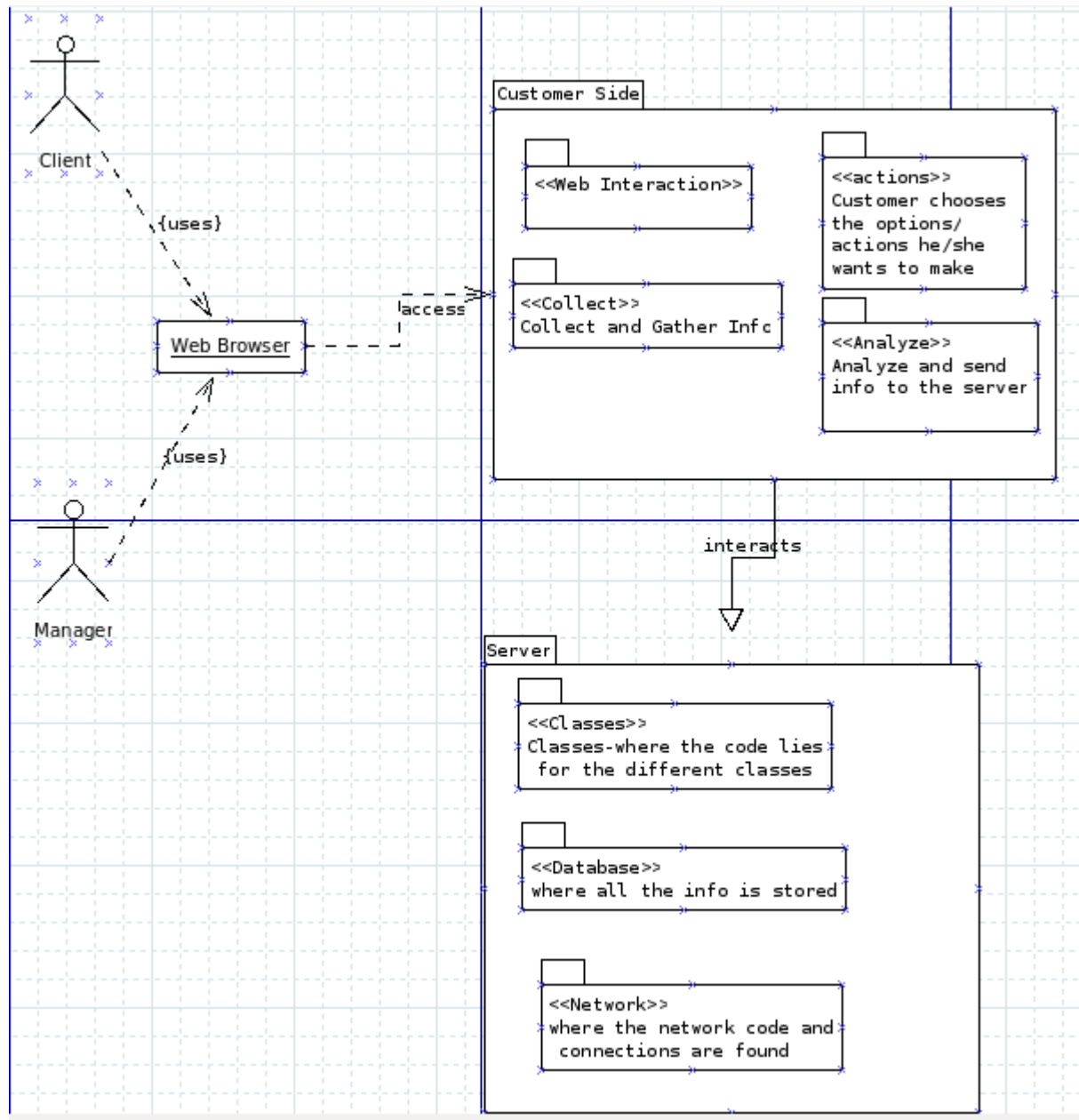
## Identifying Subsystems

Our system will make use of many existing software packages and libraries. The following list describes in short the different packages we will use in implementing our system.

- Kohana PHP framework
- Kohana ORM module (object relational mapping)
- Kohana Validation module (validates a plethora of data, from emails to credit cards)
- PayPal, Google Checkout (accept payments from our customers)
- Kohana Auth module (used to login/logout and keep track of customers and administrators)
- jQuery Javascript library
- jQuery Calendar plugin (used for easy reservation scheduling)
- Google Maps plugin (used to locate nearby garages)
- Bluetrip CSS framework

Since our system is to going to be built as a website/web service, we will need to design both the server side and client side(s) that will make up our system. The end user will interact with the main server using any standard web browser using a standard HTTP connection. The client side will be implemented in HTML/CSS/JavaScript, while the server side will be implemented in PHP. If time permits, we also plan to add other clients, specifically as mobile applications on mobile phones.

The diagram below shows the UML package diagram for our system. It is divided into two sub-systems. The first one is the client side, which through the use of a browser chooses different actions and then this sub-system sends the info to the server. The server, which is the other sub-system, is where the code for all the classes is located. It also contains the database with all the info of the customer, rates, and network settings (with other garages).<sup>[3]</sup>



## Mapping Subsystems to Hardware

The system can be broken into three main parts: a web server, a client terminal.

### Web Server

The web server runs the majority of the code, maintains a relational database concerning details of the parking garage and its reservations, and accepts input from a client-side system that allows reservations to be created and user accounts to be managed.

The server stores three main types of data: garage information, reservations, and user accounts. The garage information are things such as garage capacity and pricing structure for the garage. The reservations is the reservation records made by customers, either in advance or on the spot as walk ins. Finally, the user account info contains usage history for each customer, and a log of user activity on the system, which can be used to determine discounts for particularly well behaved customers.

### Client Side

The client side is much simpler. Basically, it consists of a web browser capable of executing HTML and some CSS. Most of the elements on the client side are HTML forms, which harvest data from the user and relay it to the web server which handles all processing and data manipulation.

In the future, it will be possible to have the client side also run as native Android and iOS apps. That functionality will be similar to the web browser based system, just with a nicer interface on top. However, the functionality of harvesting data from a user and relaying it to the server will remain the same.

## Persistent Data Storage

Our system requires us to keep track of several entities. We need to keep track of

- Customers
- Customer reservations
- Customer vehicles
- Customer billing information
- Garages (which implement our system)
- Garage price plans
- Garage administrators, and their privileges

We will be using a relational database(s), MySQL. We chose MySQL because of its reputation, its price, and because its open source software that we can alter to fit our needs if need be.

Table of contents
-------------------

1 admins_garages	Page number: {02}
2 billing_info	Page number: {03}
3 garages	Page number: {04}
4 price_plans	Page number: {05}
5 reservations	Page number: {06}
6 roles	Page number: {07}
7 roles_users	Page number: {08}
8 user_tokens	Page number: {09}
9 users	Page number: {10}
10 users_vehicles	Page number: {11}
11 vehicles	Page number: {12}
13 Relational schema	Page number: {13}

## Park-a-lot

### 1 admins\_garages

Table comments : Join table for admins and the garages they have rights to.  
Creation: Mar 07, 2011 at 09:13 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
user_id	int(10)	UNSIGNED	No			users -> id		
garage_id	int(10)	UNSIGNED	No			garages -> id		

## Park-a-lot

### 2 billing\_info

Table comments : Billing information for each customer.  
Creation: Mar 07, 2011 at 09:14 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No			users -> id		
cc_number	bigint(16)	UNSIGNED ZEROFILL	No				Credit Card Number	
address	varchar(80)		No					
city	varchar(60)		No					
state	char(2)		No					
zip_code	mediumint(5) )	UNSIGNED ZEROFILL	No					
date_added	int(10)	UNSIGNED	No					
active	tinyint(1)		No				Whether this is the card to bill at the end of each billing cycle. Only one credit card can be active at any moment.	

## Park-a-lot

### 3 garages

Table comments : Garages that use our system.  
Creation: Mar 07, 2011 at 09:17 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
address	varchar(80)		No					
city	varchar(60)		No					
state	char(2)		No					
zip_code	mediumint(8) )	UNSIGNED ZEROFILL	No					
date_added	int(10)	UNSIGNED	No					

## Park-a-lot

### 4 price\_plans

Table comments : Price plans each garage has.  
Creation: Mar 07, 2011 at 09:14 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
garage_id	int(10)	UNSIGNED	No			garages -> id		
registered_price	float	UNSIGNED	No				Price registered customers pay.	
walk_in_price	float	UNSIGNED	No				Price walk in customers pay.	
discount_rate	float	UNSIGNED	Yes	NULL			The rate at which registered customers earn discounts.	
min_price	float	UNSIGNED	No				The minimum allowable price to charge any customer.	
date_added	int(10)	UNSIGNED	No					
active	tinyint(1)		No				Only one price plan can be active at any moment.	



## Park-a-lot

### 5 reservations

Creation: Mar 07, 2011 at 09:12 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No			users -> id		
garage_id	int(10)	UNSIGNED	No			garages -> id	The garage where the user will park. Can change in the event the user is relocated.	
vehicle_id	int(10)	UNSIGNED	No			vehicles -> id	The vehicle the user is requesting parking for.	
status	enum('active', 'cancelled', 'pending')		No				The current status of the reservation.	
start_time	int(10)	UNSIGNED	No				The scheduled start time.	
end_time	int(10)	UNSIGNED	No				The scheduled end time.	
time_arrived	int(10)	UNSIGNED	Yes	NULL			The actual time the user arrived.	
time_departed	int(10)	UNSIGNED	Yes	NULL			The actual time the user departed.	
date_added	int(10)	UNSIGNED	No					
last_edited	int(10)	UNSIGNED	Yes	NULL				

## Park-a-lot

### 6 roles

Table comments : Roles (privileges) users can have in the system.  
Creation: Mar 07, 2011 at 09:14 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(11)	UNSIGNED	No		auto_increment			
name	varchar(32)		No					
description	varchar(255)		No					

## Park-a-lot

### 7 roles\_users

Table comments : Join table for users and the roles they have.  
Creation: Mar 07, 2011 at 09:15 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
user_id	int(10)	UNSIGNED	No			users -> id		
role_id	int(10)	UNSIGNED	No			roles -> id		

## Park-a-lot

### 8 user\_tokens

Table comments : Extra data collected/generated each time a user logs in.  
Creation: Mar 07, 2011 at 09:16 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No			users -> id		
user_agent	varchar(40)		No					
token	varchar(40)		No					
type	varchar(100)		No					
created	int(10)	UNSIGNED	No					
expires	int(10)	UNSIGNED	No					

## Park-a-lot

**9 users**

Creation: Mar 07, 2011 at 09:15 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
first_name	varchar(32)		No					
last_name	varchar(64)		No					
email	varchar(127)		No					
password	varchar(64)		No					
registration_date	int(10)	UNSIGNED	No					
logins	int(10)	UNSIGNED	No	0				
last_login	int(10)	UNSIGNED	Yes	NULL				

## Park-a-lot

### 10 users\_vehicles

Table comments : Join table for users and the vehicles they own/use.  
Creation: Mar 07, 2011 at 09:15 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
user_id	int(10)	UNSIGNED	No			users -> id		
vehicle_id	int(10)	UNSIGNED	No			vehicles -> id		
date_added	int(10)	UNSIGNED	No					

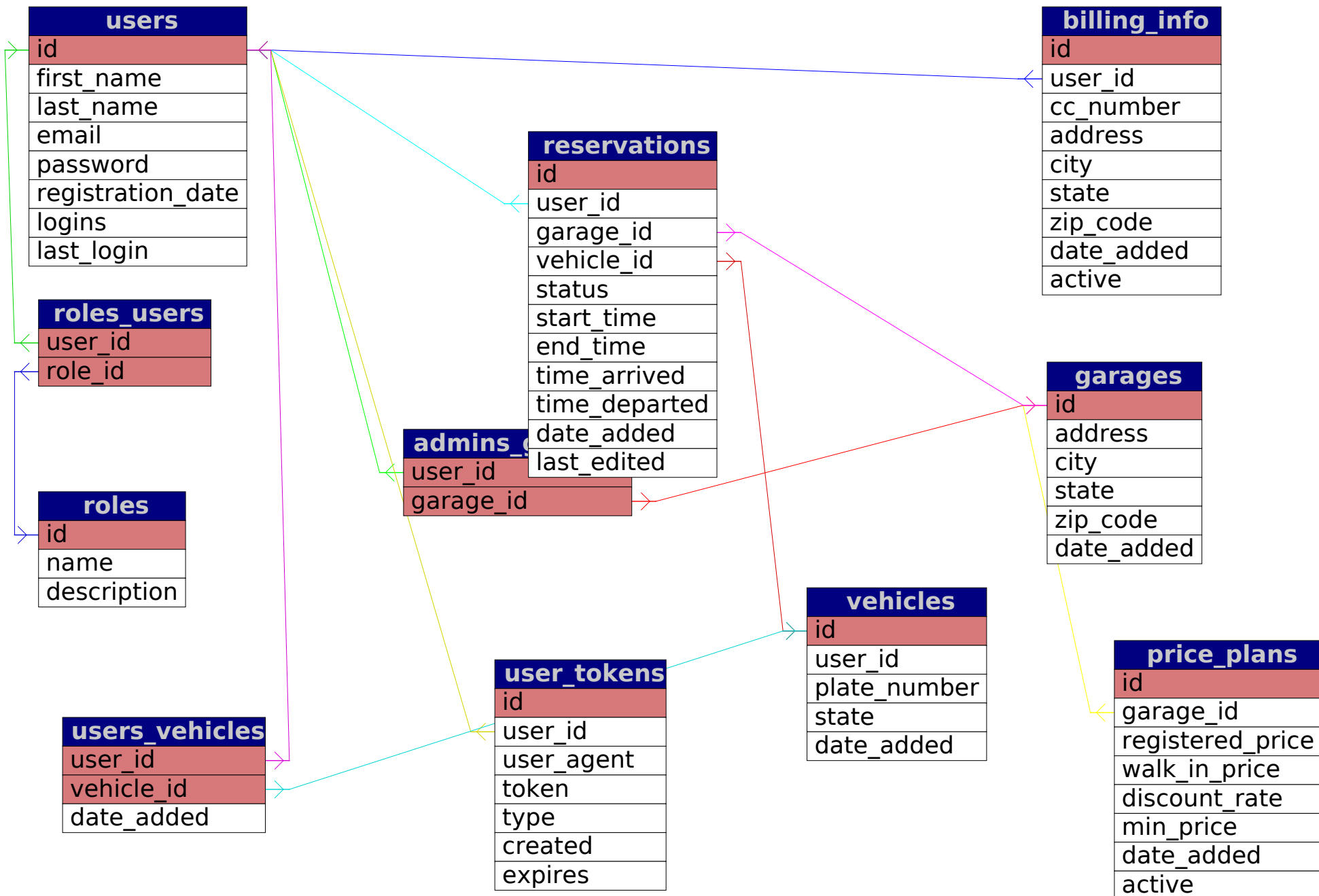
## Park-a-lot

**11 vehicles**

Table comments : User vehicles, to use with different reservations.  
Creation: Mar 07, 2011 at 09:16 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No				The user who first added the vehicle	
plate_number	varchar(10)		No					
state	char(2)		No					
date_added	int(10)		No					

## Park-a-lot





## Network Protocol

Since our system will be built as a website/web service, and thus built on a single server, there is no need for any communication protocols except the standard HTTP.

## Global Control Flow

- **Execution order** - Our software is event-driven so it depends on the customer and his purpose while interacting with the system.
- **Time Dependency** - The only timers in the system is the 15 min inactive log-in time allowed. If a customer logs into his/her account and remains inactive for 15 min then the system will logged him/her out.
- **Concurrency** - No.

## Hardware Requirements

Our system requires the following:

- **Keypad** - For the customer to enter information at the elevator.
- **Screen Display** - Minimum resolution of 640 X480 pixels. This display will be implemented in the elevator.
- **Hard Disk/Server** - 50 GB of storage space would satisfy our needs to keep track of all user data, as well as garage and reservation data. With an expectation of supporting 1M users, we would expect to amass about 5GB of data at the end of each year.
- **Sensors** - In order to figure out if the parking space is available or occupied.
- **Cameras** - In order to identify the vehicles coming in the garage and leaving.
- **Network Bandwidth** - Since there is not a huge amount of information being transferred, the amount required should be 56 kbps to access the system successfully.
- **Web browser** - In order for the client to access the software, he/she must use a semi-advance browser (i.e Mozilla Firefox 2.0 or up, Google Chrome, Internet Explorer 7 or up).

Aside from the hardware which is integrated into our system (keypad, display, etc.), the only hardware we require is disk space for our database(s). Looking at our database schema, its evident that our largest table will be the reservations table. The reservation table consists of 11 fields, 10 of which are int fields (4 bytes), and one of which is an enum field (1 byte). Each record in the table will take up 41 bytes. The amount of disk space will depend on the number of customers we intend to support, and the average rate of reservations we think each user will make and how long we will keep old records.

For example: Lets say we intend on supporting up to 100 customers, who make reservations at a rate of 1 per month. That's 1200 reservations a year. That's  $1200 * 41 = 49,200$  bytes, or 49.2

Kbytes a year. If we intend on holding reservation information for the past 3 years, that's at least 150 Kbytes before we dump out old data. Obviously, we will support a lot more customers and customers will make a lot more reservations, and we will keep data a lot longer... this is just an example. Obviously, other tables will take up data as well, but nothing is expected to grow as fast as the reservations table, so this is what we should be looking to support (I would think). In any case, disk space is cheap, bandwidth is not... so depending on the number of hits we intend on getting a day, we may need heavy processors, fast ram, etc. But, for this project, I don't think bandwidth will be much of an issue.

## Algorithms and Data Structures

### Algorithms

**Best Alternative Reservation** - this algorithm will determine the largest contiguous block of time contained in the period between a customer's desired reservation start and end times, in the event that a reservation for the entire block of time is unavailable.

For example, if a customer wants a confirmed reservation from 9am until 5pm and the creation of that reservation would overbook the garage for any amount of time within that interval, then the customer would be offered the largest contiguous block of time within that 9-5 window that would not create an overbooked garage.

Mathematically this is a very simple algorithm.

1. Determine the blocks of time during garage operating hours that the garage is not currently overbooked;
2. Filter the result of (1) to the window of time the customer requested;
3. Of the blocks of time remaining in (2), determine the largest possible block of time.

This algorithm will help the customer select a reservation. The customer may not know that their requested reservation of 9am until 5pm is unmanageable, but perhaps a 9am-4:30pm reservation is possible.

**Discount Price** - this algorithm will determine the discount registered customers accrue over time. After a customer has racked up X "perfect reservations"\* in a row, each upcoming reservation will be offered at a discount, depending on the number of previous perfect reservations.

As an example, lets assume discounts are only offered after 3 perfect reservations, the regular hourly price of a reservation is \$10/hour, the discount rate is 10% off of the previous price, and the minimum allowable price is \$6/hour.

If Bob has had 3 perfect reservations, his 4th reservation will be offered at a discount price of 10% off (\$9/hour). If Bobs 4th reservation is also perfect, then his 5th reservation will be offered at a discount price of 10% off the previous price he paid (\$8.10/hour). The trend will continue until Bob has a mishap (imperfect reservation) in which case he starts over and pays the regular fee, or until he reaches the minimum price allowed (\$6/hour) in which case he will continue to pay this price until a mishap.

Doing this, we will ensure that our customers strive for perfection with their reservations, as well as ensure that more customers register for an account as opposed to walk in.

\* In this case, a perfect reservation is one where the customer arrives and departs on time.

## **Data Structures**

Our system does not use many complicated structures (hash tables, linked lists or trees) but we do use arrays in order to quickly analyze and refer to the data in our database. The reason why we chose arrays over any other method is because it is simple to use and linked lists are not implicitly supported by PHP, the language we will be coding our server side application in.

## User Interface Design and Implementation

At this point, there were no major changes made to the user interface. We will add/edit the user interface as we define further needs, while maintaining our main goal for the user interface which is a great and easy user experience for the end user. A great user experience is rather subjective, so to test whether we achieve our main goal, we plan to have random users rate their user experience.

Since there were no major changes, please refer to Report #1 for information and screen mock-ups of the user interface. The screen mock-ups in Report #1 were actually implemented at <http://www.park-a-lot.vacau.com/> so there is no reason to re-iterate those images here. The design is meant to reduce user effort to a minimum, by providing a sleek and reduced graphical interface that is simple to understand (no screen clutter or extraneous information).

Our interface follows the “Ease-of-use” guidelines very well, as evidenced by:

1. The simple yet straightforward interface;
2. The addition of a Google Map on the homepage to help users locate their nearest Park-A-Lot garage;
3. Simple navigation on the top-right corner of the web page, with very clear names such as “Register” and “Reserve”.

A simple navigation combined with a clearly defined form on each page defines the essence of our user interface.

# Progress Report and Plan of Work

## Progress Report

Since the last report, we have managed to finalize many details of the system architecture and interactions, through the creation of the system interaction diagrams earlier in this report. These interactions are the key to fully defining our system, since they reflect the most in-depth look at how our system is expected to function.

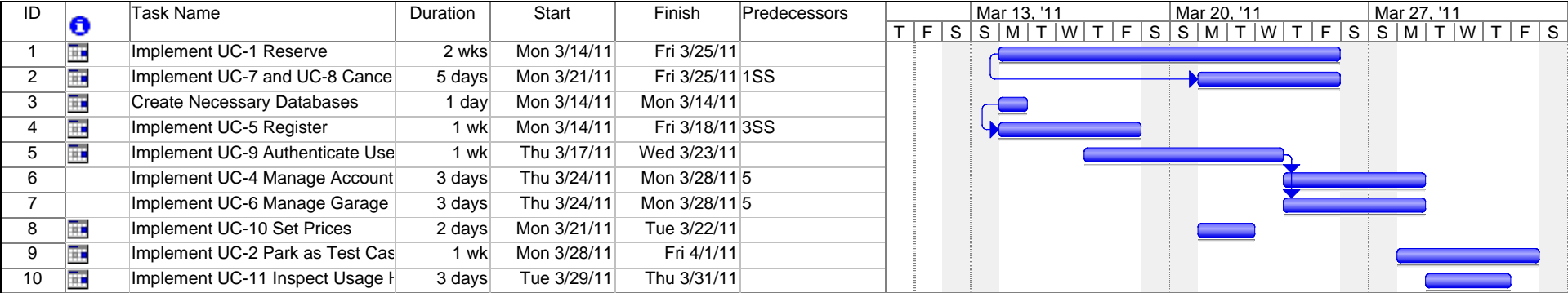
So far, no use cases have been implemented in code. However, the next three weeks of our project is dedicated to accomplishing that. The Gantt chart in the next section (Plan of Work) details how we plan to accomplish that implementation, including the order in which we will be implementing our use cases. It is worthwhile to note that since this system will not be put immediately into practice, and we do not have a physical parking garage to test it on, we will be building test cases to simulate the customer interactions of UC-2 Park.

The main functional part is our user interface, which is implemented at <http://www.park-a-lot.vacau.com>. However, there is no logic or function yet built in. It is simply a shell that will eventually house our code.

## **Plan of Work**

The Gantt chart attached to this section details how and in what order and timeline we will complete the implementation of our use cases.

The chart contains all of our project milestones, mostly use cases we will be implementing. Responsibilities for the Gantt chart are detailed in the next section, Breakdown of Responsibilities.



Project: Project1  
Date: Fri 3/11/11

Task

Split

Progress

Milestone

Summary

Project Summary

External Tasks

External Milestone

Deadline

## Breakdown of Responsibilities

Below, there is also a responsibility matrix, which assigns specific responsibilities on the Gantt chart to team members who will be completing them.

	<b>Abdul</b>	<b>Matt</b>	<b>Eric</b>	<b>Luke</b>	<b>Juan</b>
UC-1 Reserve	X	X	X		
UC-7 and UC-8 Extend/Cancel Reservation			X	X	
Create Databases	X				
UC-5 Register			X	X	X
UC-9 Authenticate User	X	X			
UC-4 Manage Account				X	X
UC-6 Manage Garage				X	X
UC-10 Set Prices		X			
UC-2 Park Test Cases		X	X		
UC-11 Inspect History	X				X

**Integration Coordination** - This will be coordinated by Matt and Abdul. It involves bringing all of the separate modules together and ensuring end-to-end connectivity and correctness.

**System Testing** - This will be performed by all group members, since it will require extensive testing for the varied possibilities of cases. There are many possible scenarios where things may go wrong, and it will be best to have each team member testing the system with varied inputs to ensure the highest quality.



## References

- [1] “Event Driven Architecture”, Wikipedia, [http://en.wikipedia.org/wiki/Event-driven\\_architecture](http://en.wikipedia.org/wiki/Event-driven_architecture)
- [2] Model-View-Controller Tutorial, <http://net.tutsplus.com/tutorials/other/mvc-for-noobs/>
- [3] Introduction to UML 2 Package Diagrams,  
<http://www.agilemodeling.com/artifacts/packageDiagram.htm>
- [4] The Kohana Framework, <http://kohanaframework.org/>