The Bike Shop

Advanced Object Oriented Design Final Project

Kevin Desmond
Steven Gennaoui
Jack Myers
Curtis White

17 December 2013

Introduction

This document represents the evolution of system design of a Bike Shop application from the analysis phase to the design phase.

The analysis phase consists of the following documents:

- **Use Case Descriptions** to describe the business flows, including primary and alternate flows
- **Use Case Diagrams** to represent system components, actors, and how they interact.
- **Analysis Classes** to depict the classes from a controller view
- **Sequence Diagrams** to depict the general flow between the various components (which generally translate to object-oriented classes)
- **Initial Functional Tests** based off the primary and alternate flows, which can be expanded on in the testing phase of the project.

The design phase consists of the following documents:

- **Updated Sequence Diagrams** to depict the general flow between the various components (which generally translate to object-oriented classes)
- **State Diagrams** which show various components and the states through which they progress
- **Design Summary** which describes the design approach, patterns used, and documents design decisions
- **Class Diagram** which shows the classes needed and key class properties and methods
- **CRC Cards** which indicate the responsibilities and collaborators of each class

Design is an iterative process. The analysis documents represent a point-in-time understanding of the Bike Shop application. Sequence diagrams have been updated to reflect design phase changes. As described in the Design Summary, there were several minor design shifts such as changing names of classes for consistency, promoting subcontrollers to controllers, etc. All of these changes will be reflected in the design documents.

There were also three major changes:

- The introduction of console login;
- The use of the State pattern for rental logs;
- The introduction of queues.

To a limited extent, these major changes will result in either modifications to analysis documents or the generation of new analysis documents. Modifications will be denoted by red text. New documents will be labeled as such.

Analysis Documents

USE CASE DESCRIPTIONS

Rent Bike Use Case

Customers have the option to browse or search the database at a customer kiosk to select available bike(s) for rental and specify the rental period for each of the selected bikes. Alternatively, they can have an employee of the bike shop browse or search the database, make customer suggestions with which the customer can agree or disagree, and record the respective rental periods. The selected bikes are then physically examined by an employee to ensure their suitability for rental. If any bikes are unsuitable, they are marked for repair (see Update Bike use case) and alternative bike selections are made by either the customer or employee. Then the system will display to the employee any special discounts that might apply to frequent customers. The employee informs the customer of the final price (including all bike rates, rental durations, rental deposits and discounts) for all bikes rented, accepts the payment for the rental and authorizes the rental. The employee will check out the bikes by either associating the bike(s) with the pre-existing customer record, or by creating a new customer record and then associating that record with the bike(s). A receipt will be printed for the customer which lists the details and cost of each bike rental. A rental request may be cancelled at any time during this process, up until the bikes are associated with the customer. After that, the Return Bike use case will apply.

Primary Flow "Typical rental" (consists of subordinate use cases)

1. Employee Selects Bikes
2. Employee Selects Customer
3. Employee Confirms Reservation

Alternate Flow "Customer selects bikes rental" (consists of subordinate use cases)

1. Customer Selects Bikes
2. Employee Selects Rental Log
3. Employee Selects Customer
4. Employee Confirms Reservation

Alternate Flow "Cancellation after bike selection" (consists of subordinate use cases)

1. Employee Selects Bikes
2. Cancel Reservation

Alternate Flow "Cancellation after bike selection" (consists of subordinate use cases)

1. Customer Selects Bikes
2. Cancel Reservation

Alternate Flow "Cancellation after rental log selection" (consists of subordinate use cases)

1. Customer Selects Bikes
2. Employee Selects Rental Log
3. Cancel Reservation

Alternate Flow "Cancellation after customer selection" (consists of subordinate use cases)

1. Employee Select Bikes
2. Employee Selects Customer
3. Cancel Reservation

Alternate Flow "Cancellation after customer selection" (consists of subordinate use cases)

1. Customer Select Bikes
2. Employee Selects Rental Log
3. Employee Selects Customer
4. Cancel Reservation

Select Bikes Use Case

Bikes may be selected from either a Customer Console or an Employee Console. Users search for bikes by parameters or browse by category. Users select bikes and enter in the rental dates for which they would like to have this particular bike. Users may select multiple bikes across multiple searches before confirming the selections and durations for those selections. Once the user is finished, a summary of their selections will be displayed at which point the selections can be finalized. After the user has confirmed their selections, a Rental Log is created (See *Create Rental Log Use Case*). If this bike selection was done at a Customer Console, the Rental Log number will be displayed to the customer and wait for confirmation that they've seen it. After they confirm it, the Rental Log will be sent to the Employee Console, so the employee knows a customer is ready to be helped. If this bike selection was done at the Employee Console, then the customer is already being helped.

Note: "Browsing" is type of search based on using categories as parameters.

Primary Flow "Employee selects bikes for customer"

1. Employee enters in search parameters to the employee console for the customer
2. The system will use the search parameters to query the database for all bikes matching the parameters.
3. The employee console will display the results of the search to the employee
4. The employee will select bike(s) and add rental data for each bike selected
5. The system will keep track of all selected bikes
6. Employee may optionally perform more search & select operations adding to this list of bikes
7. Employee will finalize the selections and rental dates
8. The system will create a rental log consisting of the bikes and rental data

Alternate Flow: "Customer selects bikes prior to approaching employee"

1. The Customer enters in search parameters to the customer console
2. The system will use the search parameters to query the database for all bikes matching the parameters
3. The customer console will display the results of the search to the customer
4. The customer will select bike(s) and add rental data for each bike selected
5. The system will keep track of all selected bikes
6. Customer may optionally perform more search & select operations adding to this list of bikes
7. Customer will finalize all selections and rental dates
8. The system will create a rental log consisting of the bikes and rental data (per Create Rental Log Use Case)
9. The customer console will display the rental log number and wait for customer acknowledgement of the number
10. The rental log will be sent to the employee console to alert an employee to a ready customer.

Create Rental Log Use Case

A Rental Log will be created for a provided list of bikes and associated rental dates for those bikes. This preliminary Rental Log does not yet contain customer information. This will also cause the bikes to go into a "reserved" state so that no other customer will see these bikes as available. The incomplete Rental Log will be returned.

Primary Flow "Creates a rental log"

1. After a user enters bike and rental data the system will create a preliminary rental log, which contains no customer information yet
2. The system will ensure that all bikes involved are put into a "reserved" state so they are unavailable to be selected by other users

Select Rental Log Use Case

A Rental Log will be selected from a list of provided logs on the Employee Console, or the selected Rental Log will be the log the Employee has created on the Employee console himself. The Rental Log that is selected will be completed or cancelled by the Employee at the Employee Console.

Primary Flow "Selects a rental log"

1. A log is selected from a list of rental logs on the Employee Console.
2. The log information is displayed on the Employee Console in a state where the log is ready to be updated with customer information.
3. The selected log is removed from the list on all Employee Consoles

Select Customer Use Case

Customer selection will be done only at the Employee Console. If the customer is known to be a returning customer, or if they are possibly a returning customer, the employee will search for the customer by some set of parameters (last name, phone number, email, etc). Existing customers matching these parameters will be displayed and the employee should choose the correct one. If the correct customer is not found, the employee may do another search on different criteria. If the customer cannot be found or the customer is known to be a new customer, then the employee should create a new Customer by adding the new customer information. After the customer has been found or a new customer has been created, this customer should be selected for this rental log. If a Customer indicates that their information has changed, then the Employee may invoke the *Update Customer Use Case*.

Primary Flow "Customer is known to have rented before"

1. The Employee will enter in parameters to find the Customer's record
2. The system will use the parameters to retrieve a list of customers that match the parameters entered
3. The system will display the list of customers to the employee
4. The Employee will select the correct Customer from the list of possibilities

Alternate Flow "Customer not found"

1. The Employee will enter in parameters to find the Customer's record
2. The system will use the parameters to retrieve a list of customers that match the parameters entered
3. The system will display the list of customers to the employee

4. Employee will choose to search again by different parameters if the correct Customer record is not displayed

Note: This flow leads to either the "Customer is known to have rented before" flow or the "New customer" flow.

Alternate Flow "New customer"

1. The Employee will choose to add a new customer
2. The Employee will enter in all customer information
3. The system will create a new customer using the provided information and return the newly created Customer for use in this rental

Confirm Reservation Use Case

The Rental Log provided will be updated to include the customer indicated. This will effectively associate the bikes in the log with the customer. This should also indicate discounts that the customer is available to receive as well as any "amount due" from previous rentals that was unpaid (See Return Bike Use Case - "Customer owes more money and cannot afford it"). The Rental Log should be displayed to the employee in full detail (bikes, customer, costs, discounts, etc). The employee should finalize this rental and approve the Rental Log with his employee ID and today's date. The Rental Log should be updated and the Employee Console should indicate that this was successful. The approval should change the state of all bikes from "reserved" to "rented." A receipt should be printed for the customer indicating all information of transaction.

Primary Flow "The rental log is updated and confirmed"

1. After the employee has selected or created a customer for this rental, the rental log that was created for the set of bikes and rental data will include the customer
2. All applicable discounts as well as amounts due will be calculated and put on the rental log
3. The employee console will display the final rental log (bikes, rental dates, customer info, cost breakdown, discounts, etc)
4. The employee will finalize the rental after receiving payment
5. The system will approve the rental log, change all bike statuses to "rented" and update the database.
6. The system should print a receipt summarizing this rental log

Alternate Flow "The customer cannot pay for the rentals"

1. After the employee has selected or created a customer for this rental, the rental log that was created for the set of bikes and rental data will include the customer
2. All applicable discounts as well as amounts due will be calculated and put on the rental log
3. The employee console will display the final rental log (bikes, rental dates, customer info, cost breakdown, discounts, etc)
4. The customer cannot pay for the rentals due to lack of cash, or credit card refusal.
5. The employee cancels the reservations (see Cancel Reservation Use Case).

Alternate Flow "The customer changes their mind about selected bikes"

1. After the employee has selected or created a customer for this rental, the rental log that was created for the set of bikes and rental data will include the customer
2. All applicable discounts as well as amounts due will be calculated and put on the rental log
3. The employee console will display the final rental log (bikes, rental dates, customer info, cost breakdown, discounts, etc)
4. The customer decides to alter the bikes they are renting.

5. The employee cancels the reservations (see Cancel Reservation Use Case).
6. A new request is created and the process starts over from the beginning (see Rent Bikes / Select Bikes Use Case).

Cancel Reservation Use Case

If after bikes have been selected for reservation and a rental log has been created and before the reservation has been confirmed, the customer wishes to cancel the rental, the employee console should cancel the rental log and refresh and all bikes included in the rental log should have their statuses changed from “reserved” to “available.”

Primary Flow “Cancel Reservation”

1. While viewing the rental log on the Employee’s console, the employee cancels the reservation by pressing the Cancel button.
2. All bikes involved in the rental log should be made “available”
3. The rental log should be updated to reflect that it was a “cancelled” log
4. The employee observes that the cancellation was successful.

Return Bike Use Case

Customers will return the bike(s) to the bike shop when they are finished with them. The employee verifies that correct bikes are returned by comparing the bikes with the customer’s paper receipt or via a lookup of the rental record. If the bikes are returned late, additional charges are added by the employee. If the rented bike(s) are no longer in the possession of the customer, the customer will be charged a replacement fee and the employee indicates the loss on the bike record.*(See Assess Bike Use Case)* All returned bikes are inspected for damages by the mechanic (*See Inspect Bike Use Case*). The status of all returned bikes is set as described in the Inspect Bike Use Case. If the mechanic finds any damage, additional charges will be levied against the customer. Assuming any additional charges are less than the deposit, the customer’s deposit is returned in full or partially. If the additional charges exceed the deposit, the customer is charged separately for the balance.

Primary Flow

1. Employee enters rental information in the console (i.e. customers name, bike id, rental id)
2. The system retrieves the rental log based on search criteria from the above step.
3. Bike(s) from rental log are displayed on the console.
4. Employee selects the bike(s) being returned as well as lost or stolen bike(s).
5. Assess Bike (subordinate use case)
6. The amount due or amount to be returned is calculated by the system.
7. Logs and final amount due or amount to be returned are displayed on the console
8. Employee finalizes the return transaction and accepts payment from the customer or gives money back to the customer.
9. Rental logs are updated in the database.
10. The receipt is printed.

Alternate Flow “Customer owes more money and cannot afford it”

1. Employee enters rental information in the console (i.e. customers name, bike id, rental id)
2. The system retrieves the rental log based on search criteria from the above step.
3. Bike(s) from rental log are displayed on the console.
4. Employee selects the bike(s) being returned as well as lost or stolen bike(s).
5. Assess Bike (subordinate use case)
6. The amount due is calculated by the system.

7. Logs and final amount due are displayed on the console
8. The customer owes money and cannot pay at this time.
9. Rental logs are updated in the database.
10. The receipt is printed which also acts as a bill
11. The customer record is updated with an amount due to the bike shop.

Assess Bike Use Case

The customer will be given additional charges when bikes are returned late, or have been lost or stolen.

Primary Flow

1. The system processes any late returns and updates a list of charges.
2. Status of bikes in database is changed to indicate the bike is lost.
3. Lost bikes are queried to get the replacement fee.
4. The replacement fee is added to the list of charges.
5. Rental and bike logs are updated in the database.
6. Returned bikes are inspected (subordinate use case)
7. Damage charges are added to the list of charges.
8. All charges are returned to the Return Bike Use Case.

Inspect Bike Use Case

The mechanic will take bike(s) that are being returned and inspect them for any damage done while the customer had the bike. If the bike is still in good condition, it may be immediately made available for another customer to rent. If the bike is no longer usable due to damage or if the bike needs maintenance/tune up, the bike will be marked for repair and made unavailable for renting. Damages should be reported (*see Assess Bike Use Case*) so that associated fees can be applied to customer bills.

Primary Flow "Mechanic inspects bikes"

1. The Mechanic goes to the mechanic console and selects one of the rental logs for bikes that have returned but not inspected.
2. The Mechanic inspects one of the bikes for the selected rental log.
3. The Mechanic adds any repair charges to the bike. The act of entering repair charges will ensure the bike remains unavailable for rental once the information is stored in the database.
4. When finished inspecting and adding repair charges to all the bikes on the selected rental log, the Mechanic indicates that he is finished working on this log.
5. The System stores all of the changes to the inspected bikes to the database.
6. The System increments the charges for each bike to include damage fees on the rental log.
7. The System stores the updated rental log to the database.
8. The processed rental log disappears from the mechanic console.

Repair Bike Use Case

The mechanic is responsible for repairing all bikes. Repairs include heavy damage as well as maintenance and tune ups. When bikes are fully repaired they should be made available for a customer to rent.

Primary Flow "Mechanic repairs bikes"

1. The Mechanic physically repairs a bike.
2. The Mechanic enters search parameters to find the bike he has just repaired.
3. The Mechanic selects a bike from the list of bikes needing repairs that match his search criteria.
4. The Mechanic indicates the repair costs to the bike are now zero, which makes the bike available for rental.
5. The System stores the updated bike information to the database.

Database Transaction Use Case

Note: Database Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual types of transaction (Public Transaction, Executive Transactions, and Employee Transactions) give the features that are specific to that type of transaction.

A database transaction use case is started within a Public Transaction, Executive Transactions, and Employee Transactions when a user's actions has to request information or an action for the database to resolve. The transaction request will be sent to the database.

If the database completes the request, any information or action requests will be performed.

If a transaction is cancelled by the user, or fails for any reason, the database will be resolved to its prior state before the transaction to maintain integrity.

Public Transaction Use Case

Note: Public Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual types of transaction (Select Bike) give the features that are specific to that type of transaction.

A public transaction use case is started with a Select Bike Use Case when a user transaction has to request information that is public knowledge. The user transaction request will be sent to the database.

If the database completes the request, the information that is public will be sent back.

Cases under this abstract case do not alter the state/information of the database in anyway.

Primary Flow: "Shows Access Level of Database"

Query Bike Use Case

A query bike queries the database for a list of bikes by the bike attributes (e.g. number of seats, size of bike, etc). The list will be returned to be displayed to the user.

Primary Flow: “Query Database Access on Bike Table”

1. Sends query to database using listed bike attributes as clauses that were received from calling function.
2. Gets a List back of bikes from the database.
3. Returns list back to calling function

Employee Transaction Use Case

Note: Employee Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual types of transaction (Query Customer, Update Customer, Update Bike, Add Customer, Query Logs, Update Logs, Add Logs) give the features that are specific to that type of transaction.

A Employee transaction is started within a Query Customer, Update Customer, Update BIke, Add Customer, Query Logs, Update Logs, and Add Logs use cases. The user will be asked for appropriate information (e.g. cutomer attributes, bike information, update information, log information, etc). The employee transaction will be sent to the database, with information for the transaction to complete.

If the database completes the transaction, any information needed is sent back to the user (console). For the user to complete any task at hand.

If a transaction is cancelled by the user, or fails for any reason, the database will be set back to its state prior to the employee transaction.

Primary Flow: “Shows Access Level of Database”

Query Customer Use Case

Query Customer searches the database for a given customer by customer attributes. It returns a list of customers back.

Primary Flow: “Query Database on the Customer Table”

1. Sends query to database using listed customer attributes as clauses that were received from calling function.
2. Gets a List back of customers from the database.
3. Returns list back to calling function

Update Customer Use Case

Update Customer changes the customer attributes of a given customer in the database. It returns a true/false value stating whether the change happened. It creates a copy of the previous Customer row in the History Customer log table.

Primary Flow: “Update Database on the Customer Table”

1. Sends an update statement to the database using customer attributes that were received from the calling function.
2. Gets back a confirmation on successful change or not
3. Sends confirmation back to calling function (can be ignored or not)

Update Bike Use Case

Update Bike changes the bike attributes of a given bike in the database. It returns a true/false value stating whether the change happened. It creates a copy of the previous Bike row in the History Bike Log table

Primary Flow: “Update Database on the Bike Table”

1. Sends an update statement to the database using bike attributes that were received from the calling function.
2. Gets back a confirmation on successful change or not
3. Sends confirmation back to calling function (can be ignored or not)

Add Customer Use Case

Add Customer adds a customer to the database. It gives the customer a unique ID to represent the customer. It sets all the attributes of the customer in the database, as well.

Primary Flow: “Add to Database on the Customer Table”

1. Sends an add statement to the database using the customer attributes that were received from the calling function.
2. Gets back a confirmation on successful add or not
3. Sends confirmation back to calling function (can be ignored or not)

Query Logs Use Case

Query Logs searches the log table of the database using a particular query detail. It returns a List of logs whether empty or not.

Primary Flow: “Query Database on the Log Table”

1. Sends query to database using listed log attributes as clauses that were received from calling function.
2. Gets a List back of logs from the database.
3. Returns list back to calling function

Update Logs Use Case

Update Logs updates a log based on a query detail. It executes an update detail or delete detail to change the log in the database. It creates a copy of the previous Rental Log in the History Rental Log table.

Primary Flow: “Update Database on the Log Table”

1. Sends an update statement to the database using log attributes that were received from the calling function.
2. Gets back a confirmation on successful change or not
3. Sends confirmation back to calling function (can be ignored or not)

Add Logs Use Case

Add logs uses add detail to create a new unique log in the database.

Primary Flow: “Add to Database on the Log Table”

1. Sends an add statement to the database using the log attributes that were received from the calling function.
2. Gets back a confirmation on successful add or not
3. Sends confirmation back to calling function (can be ignored or not)

Query Details Use Case

Query detail is the attribute that is used to search for a particular log the user is looking for.

Primary Flow: “Query Database on the Detail Table”

1. Sends query to the database using listed detail attributes as clauses that were received from calling function.
2. Gets a List back of details from the database.
3. Returns list back to calling function

Delete Details Use Case

Delete detail deletes a unique detail from a log.

Primary Flow: “Delete from Database on the Detail Table”

1. Sends a remove statement to the database using the detail attributes that were received from the calling function.
2. Gets back a confirmation on successful remove or not
3. Sends confirmation back to calling function (can be ignored or not)

Update Details Use Case

Update detail changes an attribute(s) of a detail in a log.

Primary Flow: “Update Database on the Detail Table”

1. Sends an update statement to the database using detail attributes that were received from the calling function.
2. Gets back a confirmation on successful change or not
3. Sends confirmation back to calling function (can be ignored or not)

Add Details Use Case

Add details adds detail(s) to a log.

Primary Flow: “Add to Database on the Detail Table”

1. Sends an add statement to the database using the detail attributes that were received from the calling function.
2. Gets back a confirmation on successful add or not
3. Sends confirmation back to calling function (can be ignored or not)

Executive Transaction Use Case

Note: Executive Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual

types of transaction (Acquire Bike, Decommission Bike, Remove Customer) give the features that are specific to that type of transaction.

Executive transaction is started within an Acquire Bike, Decommission Bike, Remove Customer use cases. These cases are owner level access only. The owner will be asked for information critical to the transaction.

Primary Flow: “Shows Access Level of Database”

Acquire Bike Use Case

Acquire Bike add a bike to the database. It gives the bike a unique ID to represent the bike. The bike attributes are set in the database, as well.

Primary Flow: “Add to Database on the Bike Table”

1. Sends an add statement to the database using the bike attributes entered.
2. Gets back a confirmation on successful add or not

Decommission Bike Use Case

Decommission Bike removes a bike from the database fully. It releases the unique ID that represents the bike to be used in an Acquire Bike action. All attributes of the bike in database are removed as well.

Primary Flow: “Delete from Database on the Bike Table”

1. Sends a remove statement to the database using the bike attributes provided.
2. Gets back a confirmation on successful remove or not

Remove Customer Use Case

Remove Customer removes a customer from the active database. It keeps all records needed in accordance with the laws of the land for the particular number of years as stated in another database. It releases the unique ID of the customer after the records are not needed anymore, and removes all attributes of the customer as well. The customer removed will not be available in the active database in any way.

Primary Flow: “Delete from Database on the Customer Table”

1. Sends a select statement to the database to select the Customer row to be deleted from information provided.
2. Copies row into storage database with a timestamp.
3. Sends a remove statement to the database using the Customer attributes provided.
4. Gets back a confirmation on successful remove or not.

Login Use Case

The operator will have to login to the console, the customer will be automatically authenticated when logging into a customer console. The employee will need to provide credentials into the console to login.

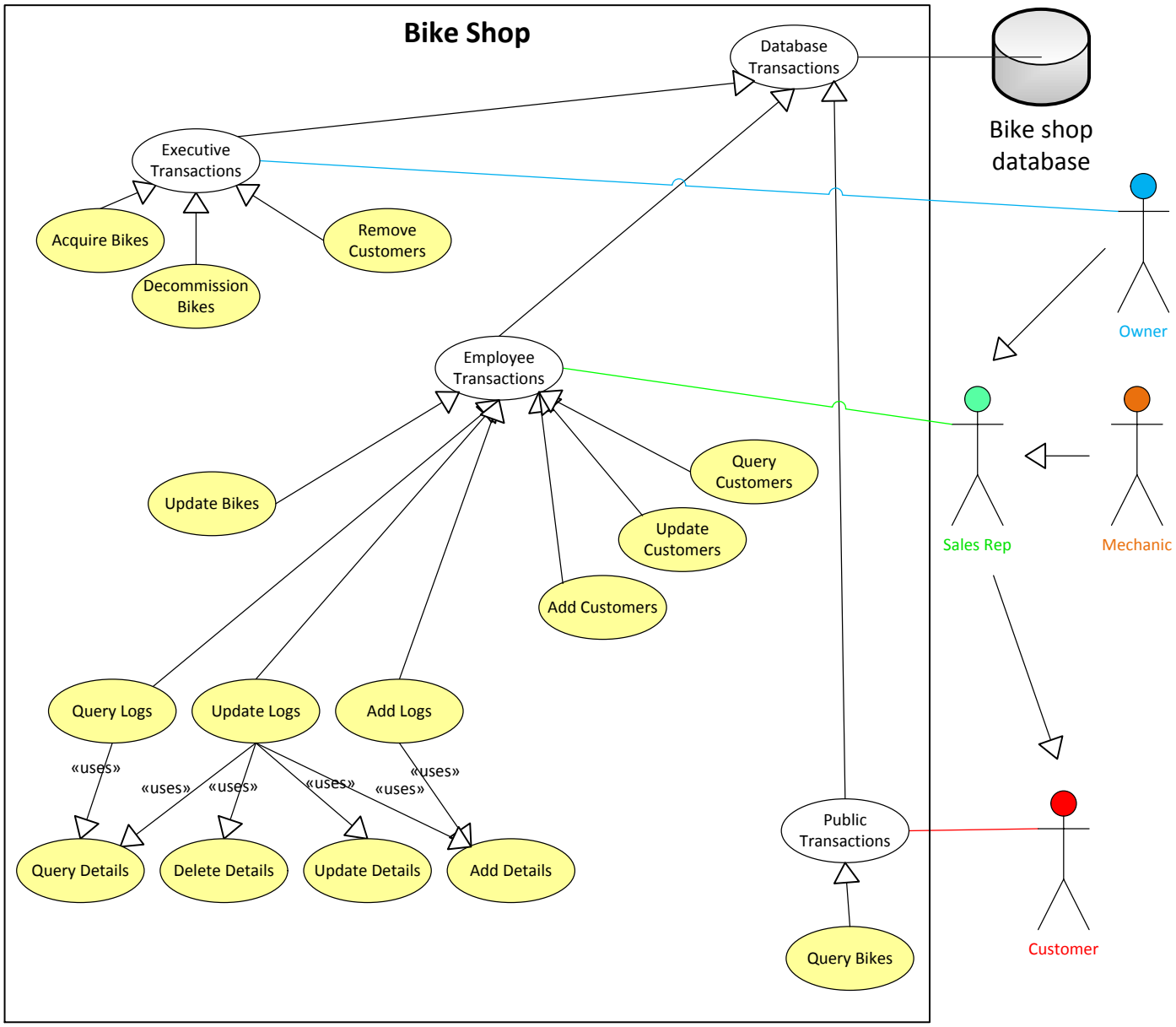
Primary Flow: "Logs into console"

1. Gets user input of password and username

2. Creates credential object with password and username
3. Authenticates credential
4. Returns true/false on authentication

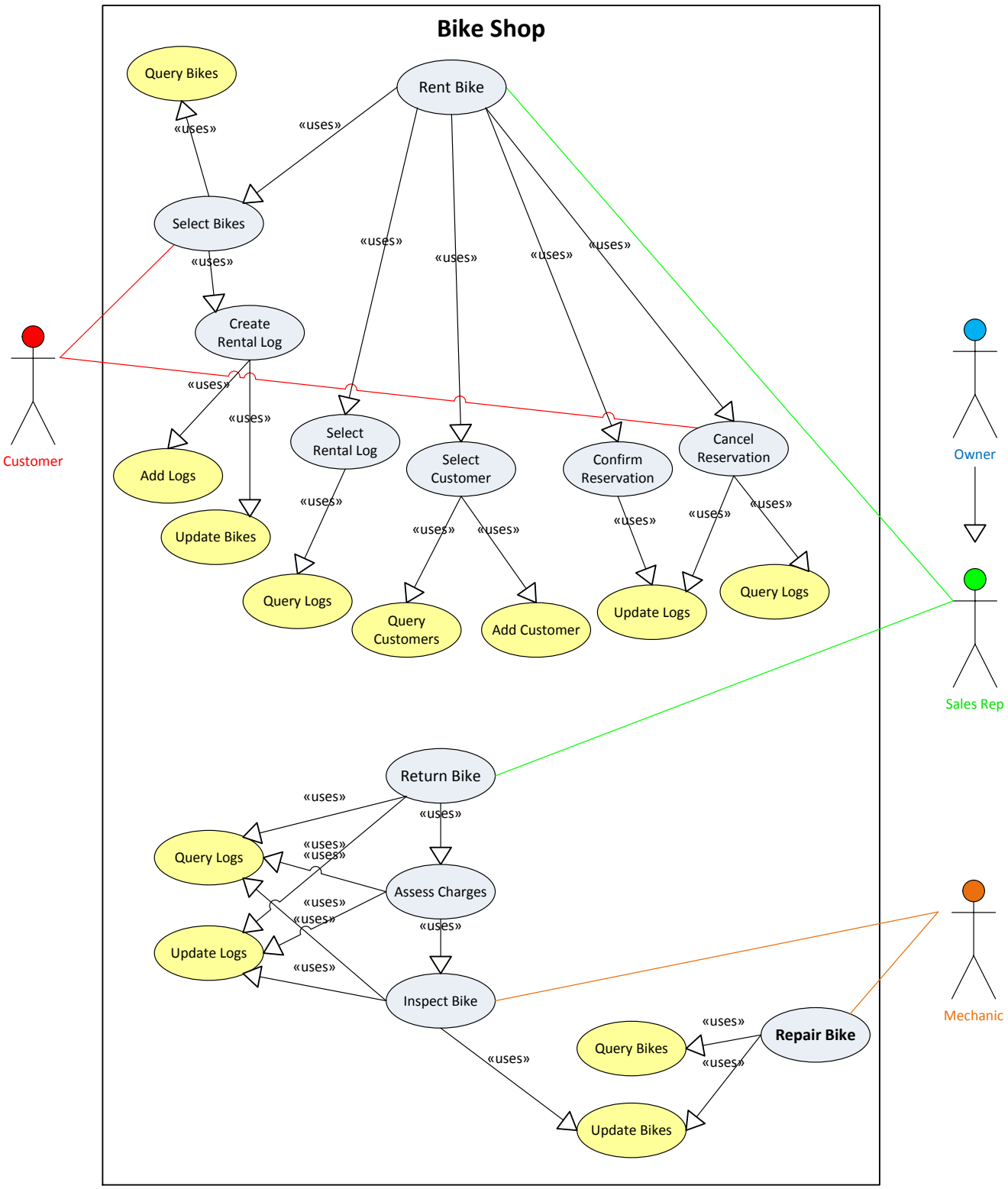
USE CASE DIAGRAMS

Ad Hoc Database Access

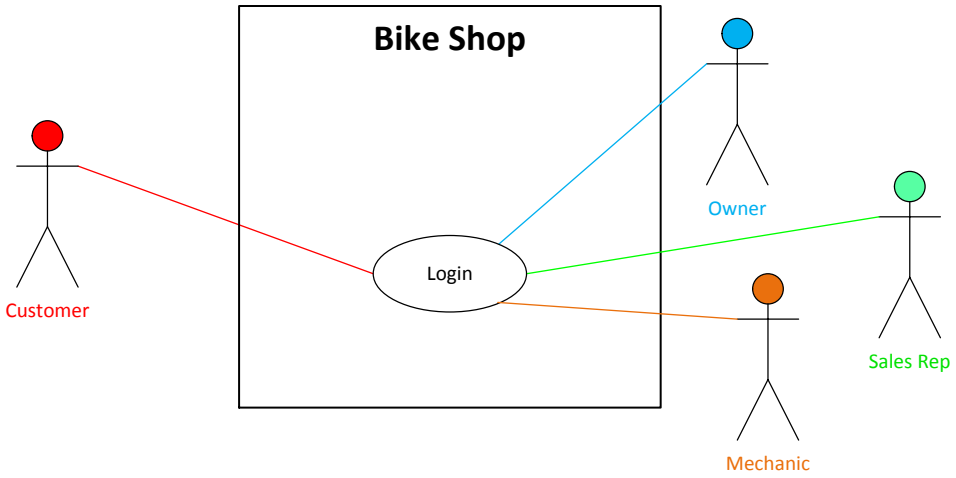


Ad hoc database transactions occur when an actor's **sole** purpose is to interact with the database; i.e., when a database transaction is not part of another use case.

Primary Use Cases

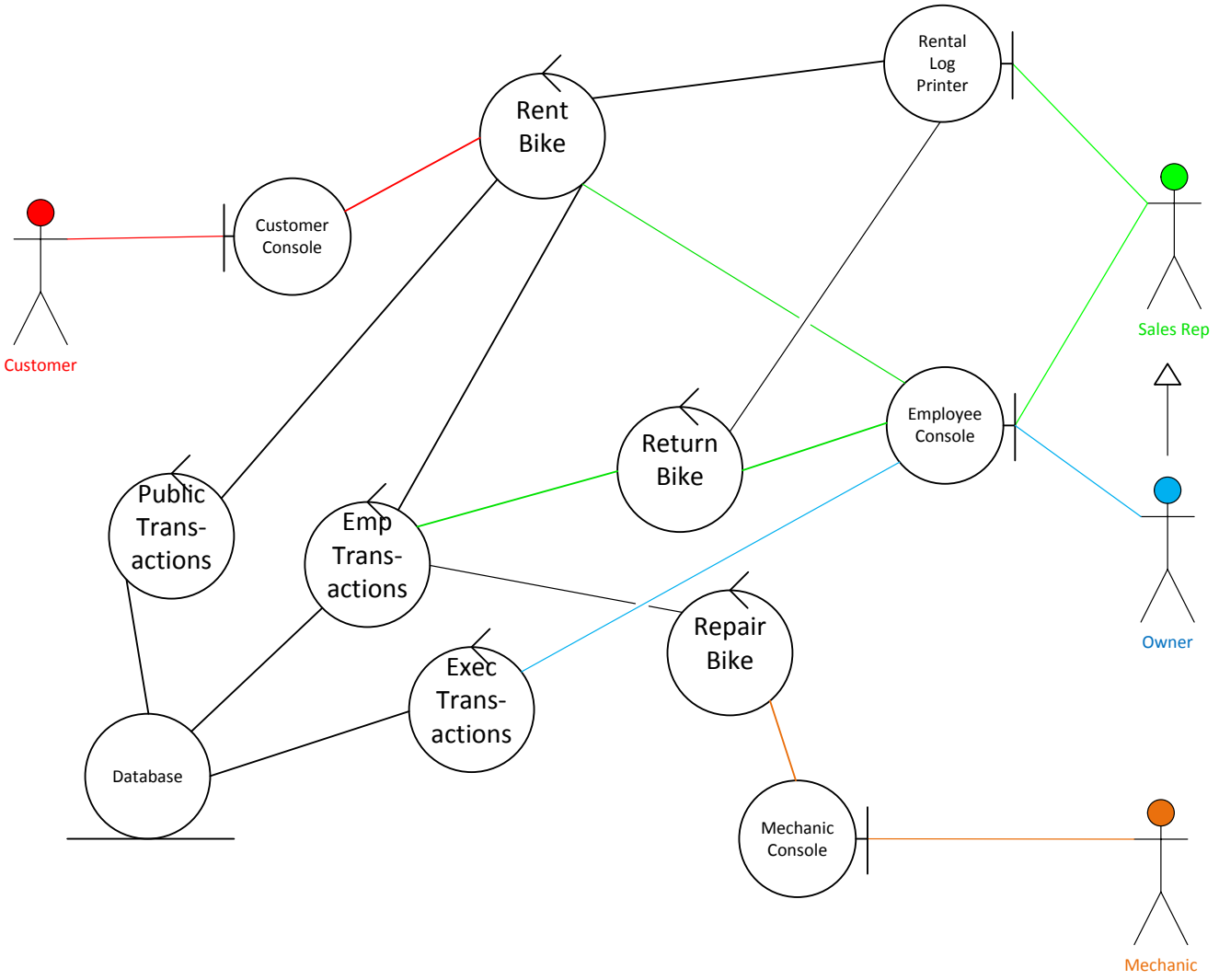


Login Use Case (new)



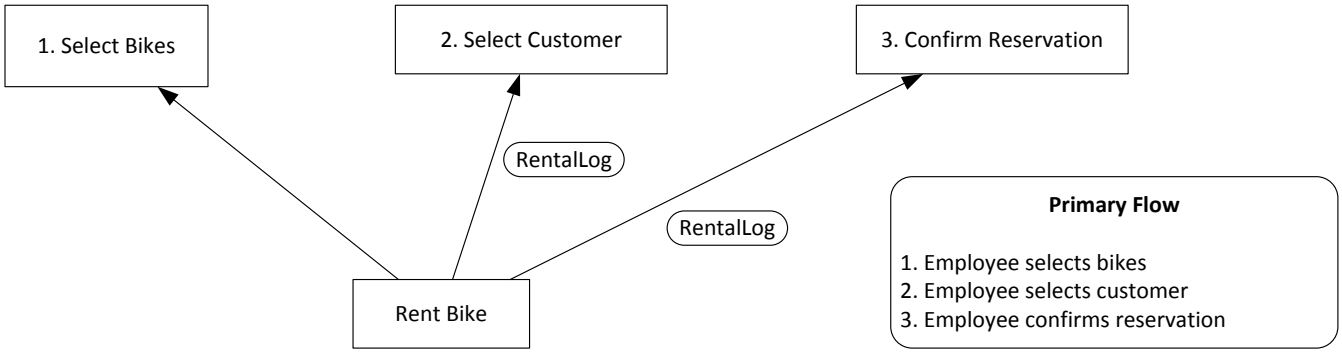
Note: Login is the only use case where actors are not able to "act as" another actor.

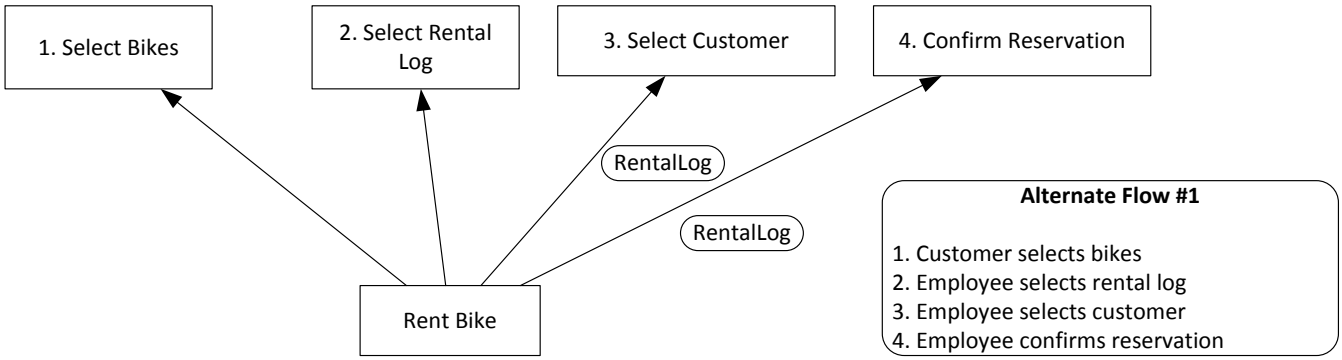
ANALYSIS CLASSES

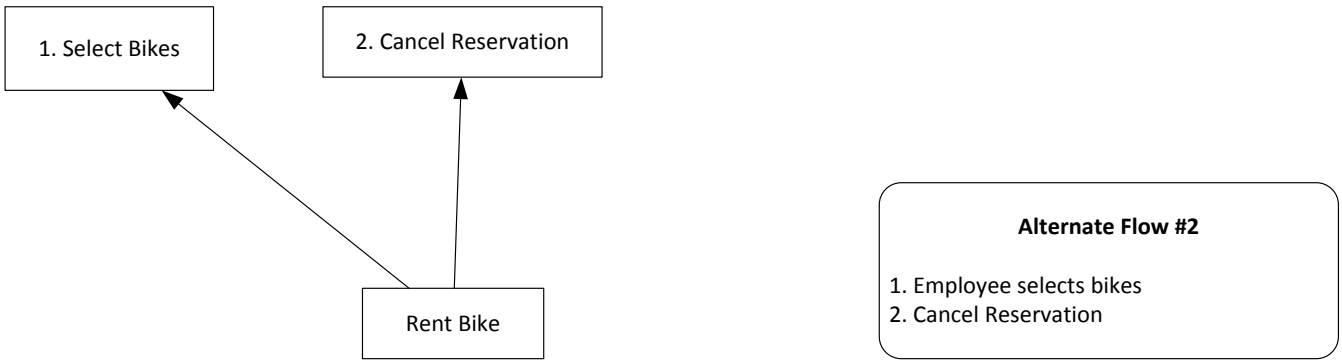


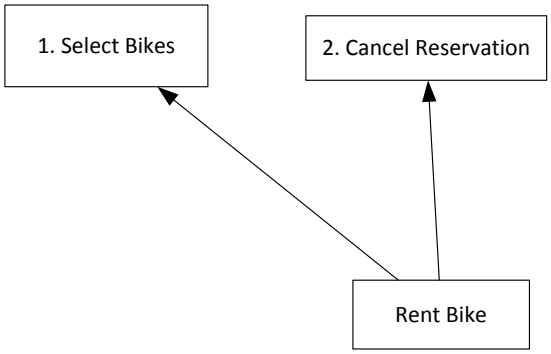
SEQUENCE DIAGRAMS

Rent Bike (Interaction Diagrams)



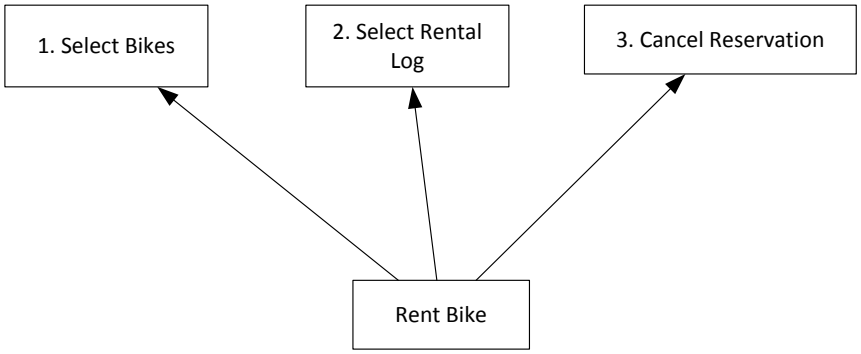






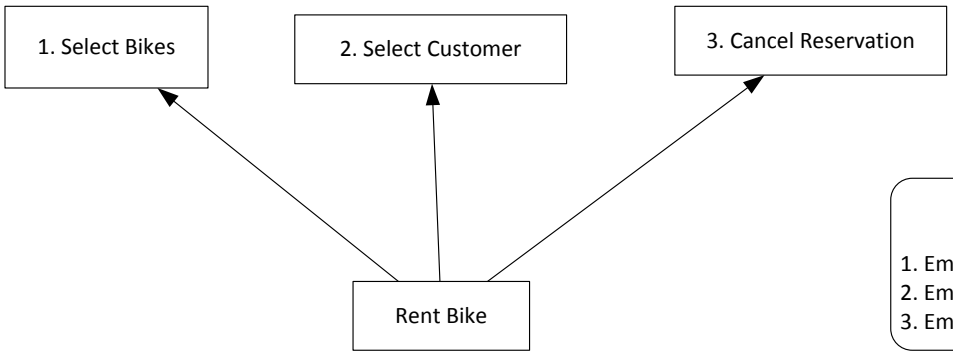
Alternate Flow #3

1. Customer selects bikes
2. Cancel Reservation



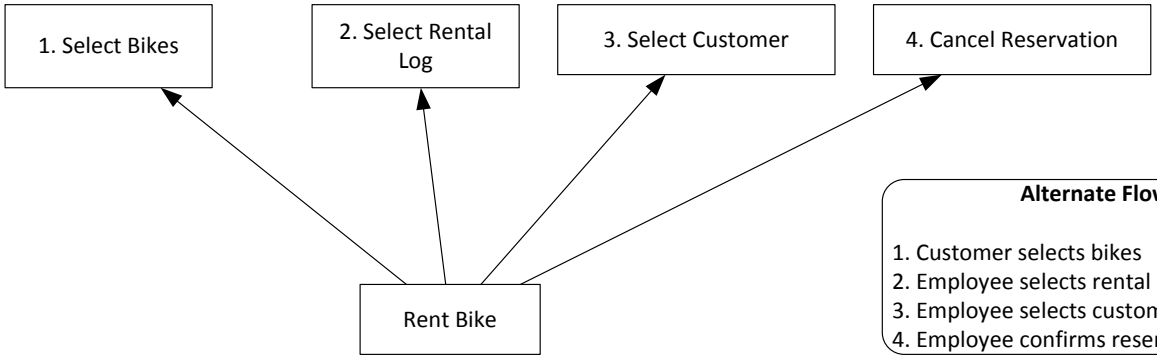
Alternate Flow #4

1. Customer selects bikes
2. Employee selects rental log
3. Cancel Reservation



Alternate Flow #5

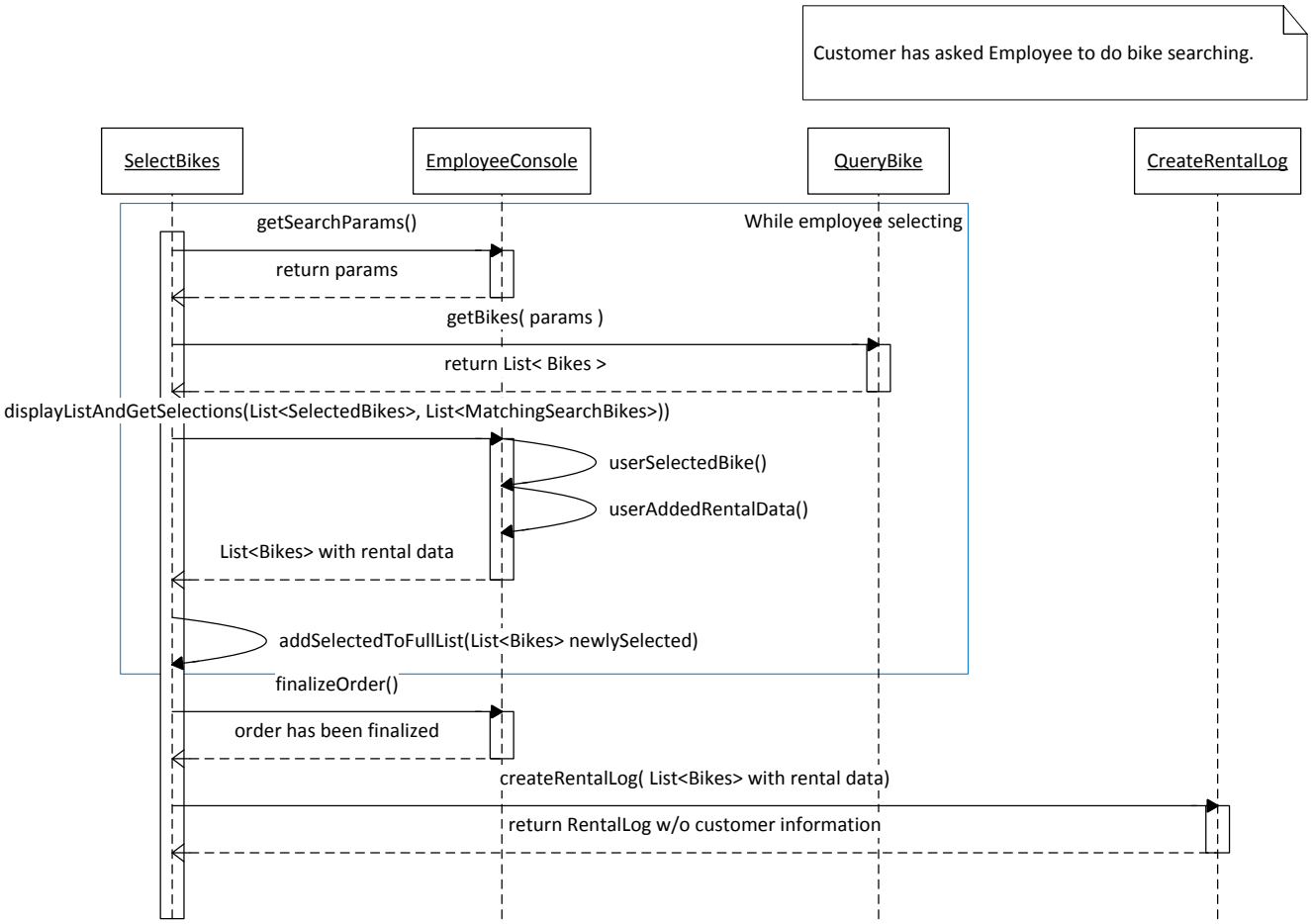
1. Employee selects bikes
2. Employee selects customer
3. Employee confirms reservation

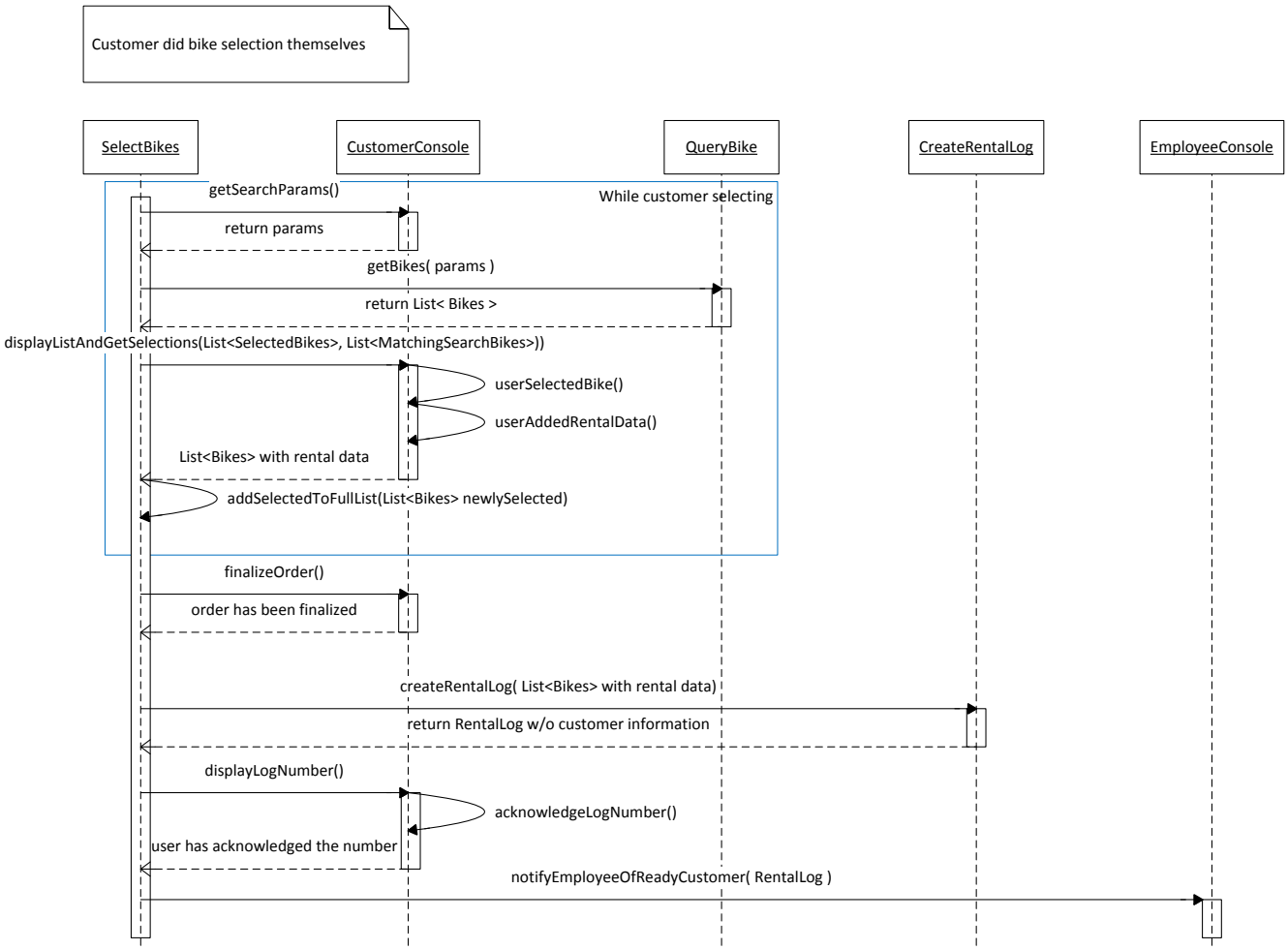


Alternate Flow #6

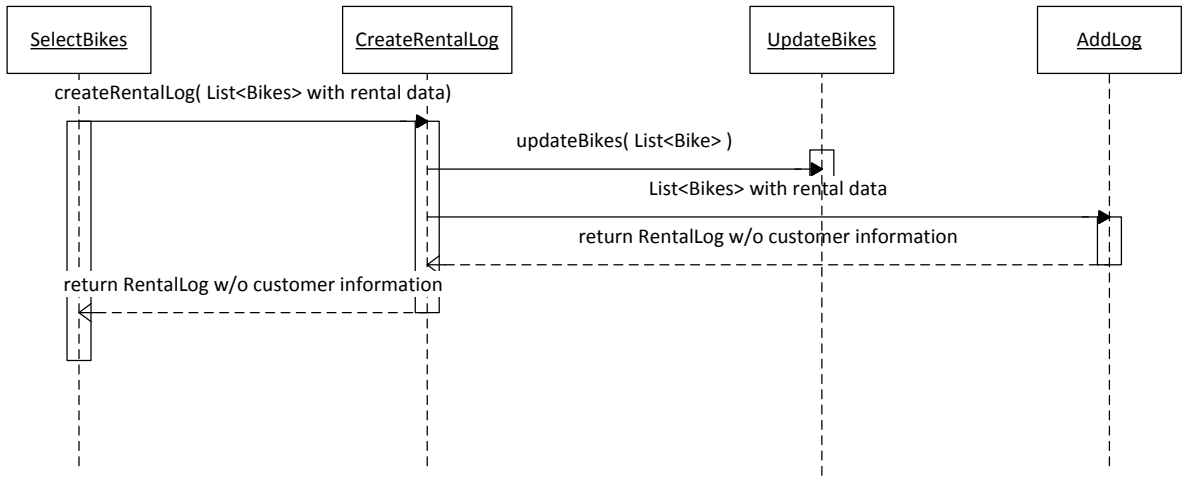
1. Customer selects bikes
2. Employee selects rental log
3. Employee selects customer
4. Employee confirms reservation

Select Bikes

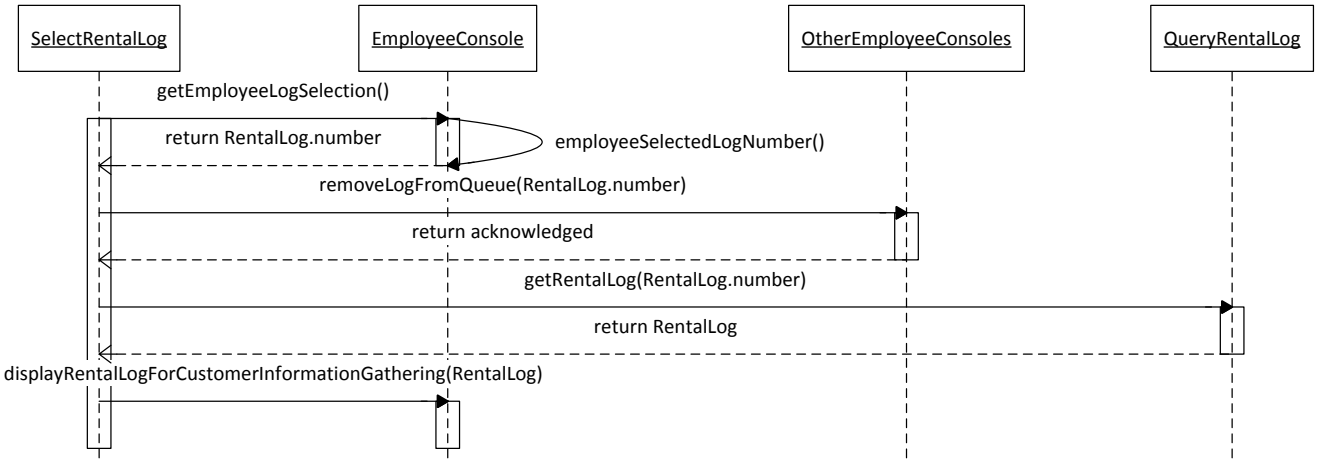



Customer did bike selection themselves
SelectBikes
CustomerConsole
QueryBike
CreateRentalLog
EmployeeConsole
While customer selecting
getSearchParams()
return params
getBikes(params)
return List< Bikes >
displayListAndGetSelections(List<SelectedBikes>, List<MatchingSearchBikes>))
userSelectedBike()
userAddedRentalData()
List<Bikes> with rental data
addSelectedToFullList(List<Bikes> newlySelected)
finalizeOrder()
order has been finalized
createRentalLog(List<Bikes> with rental data)
return RentalLog w/o customer information
displayLogNumber()
acknowledgeLogNumber()
user has acknowledged the number
notifyEmployeeOfReadyCustomer(RentalLog)

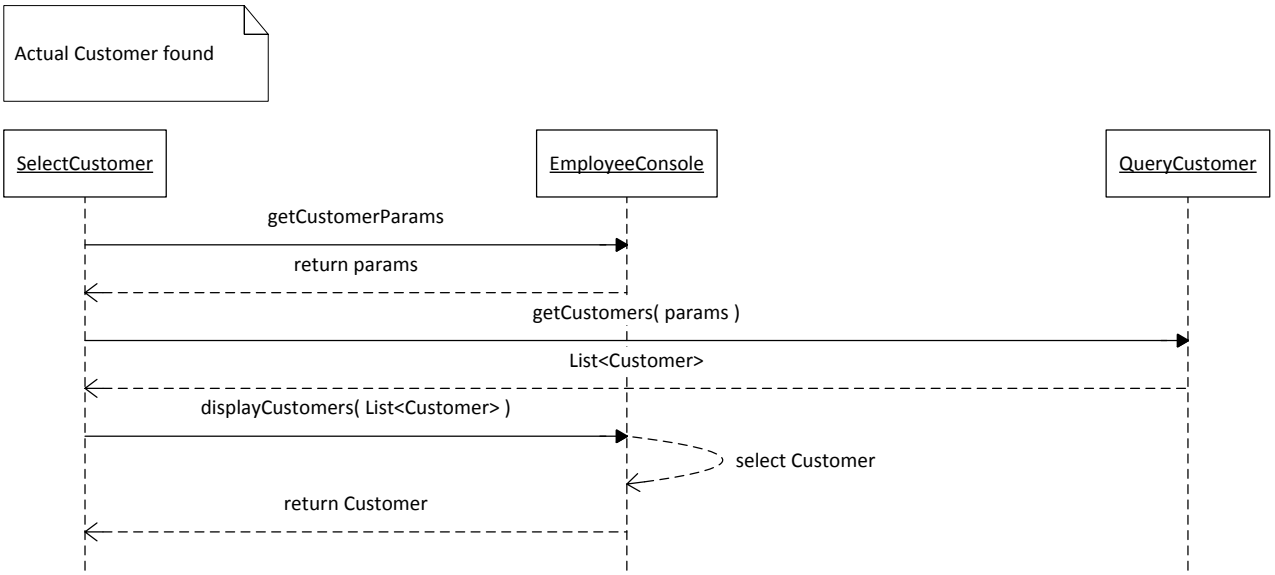
Create Rental Log

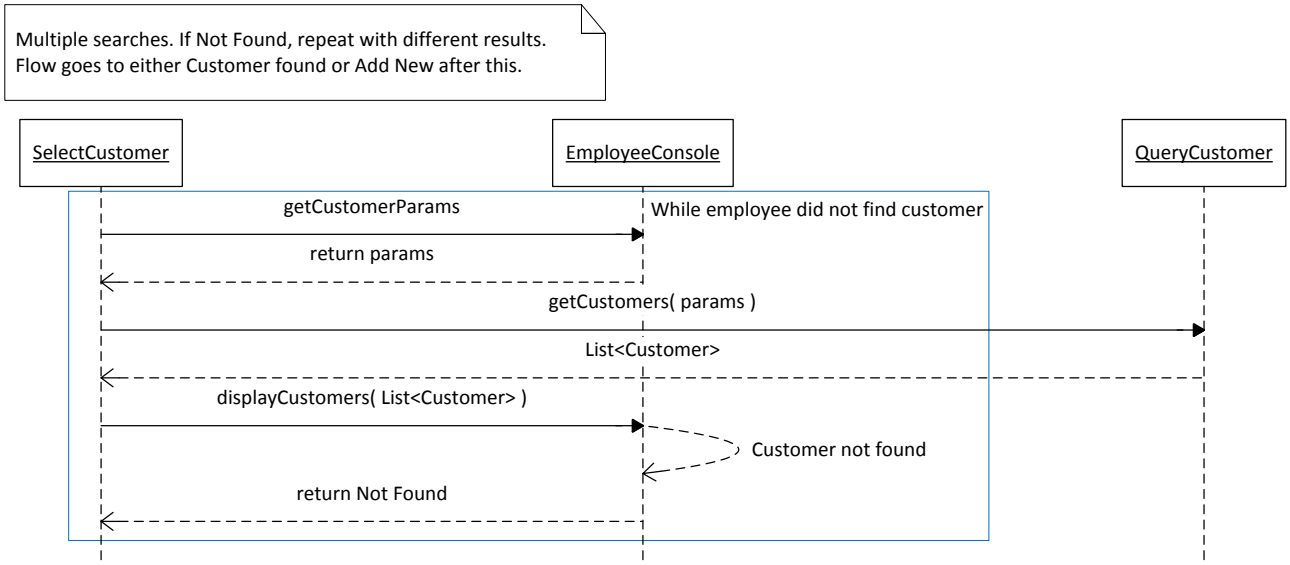


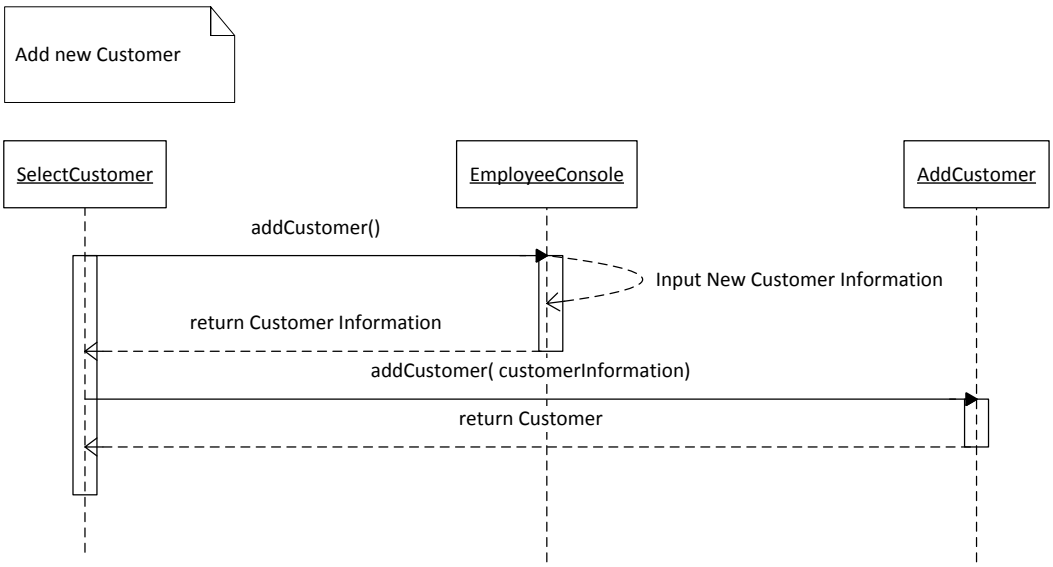
Select Rental Log



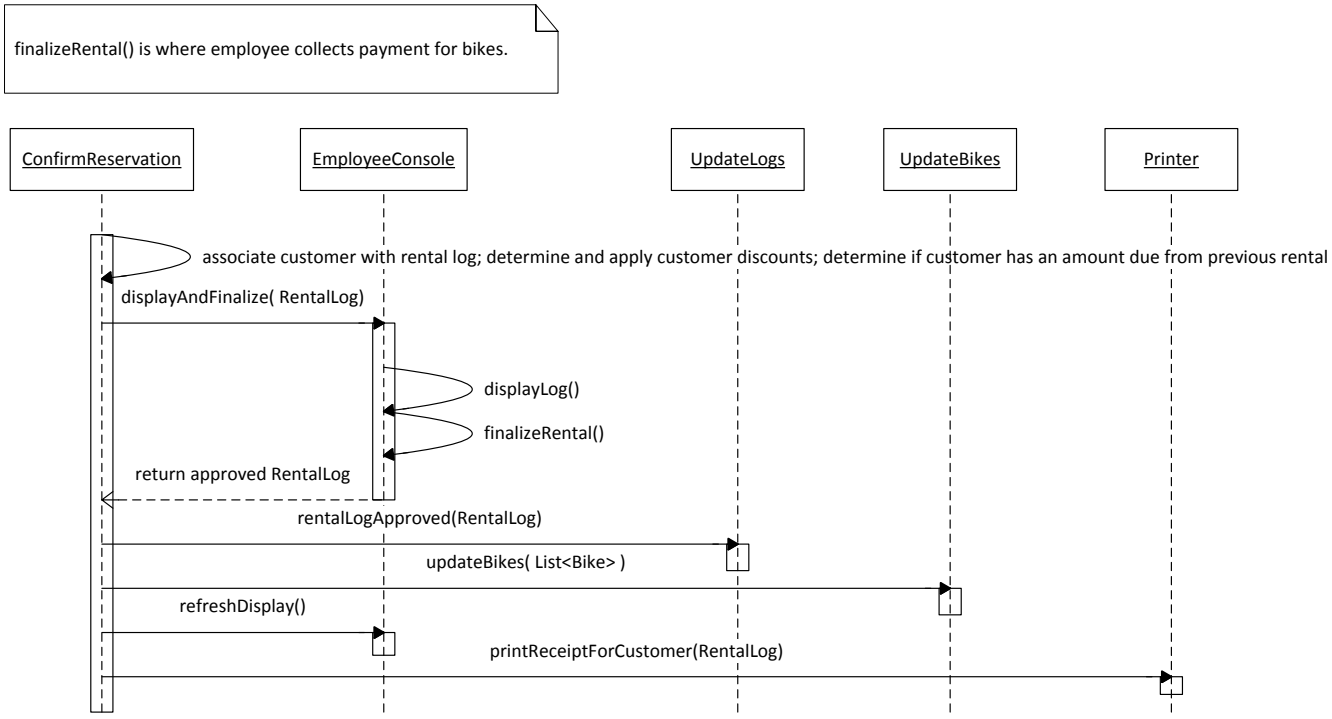
Select Customer

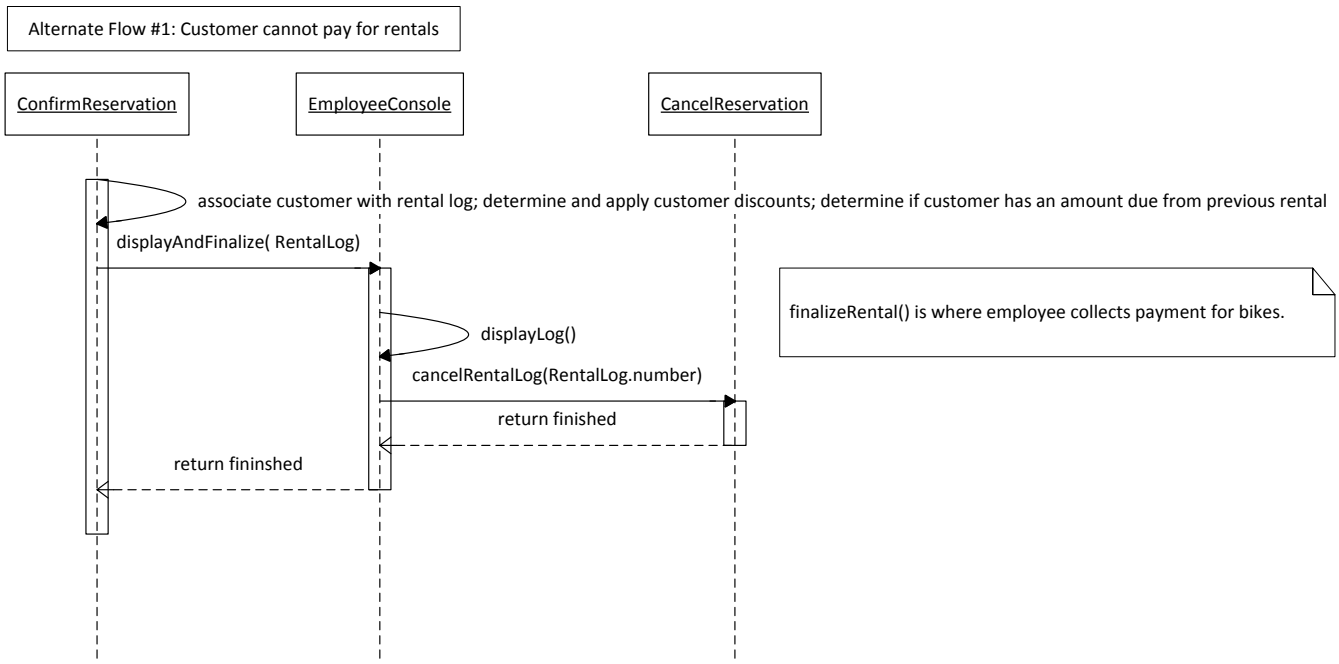


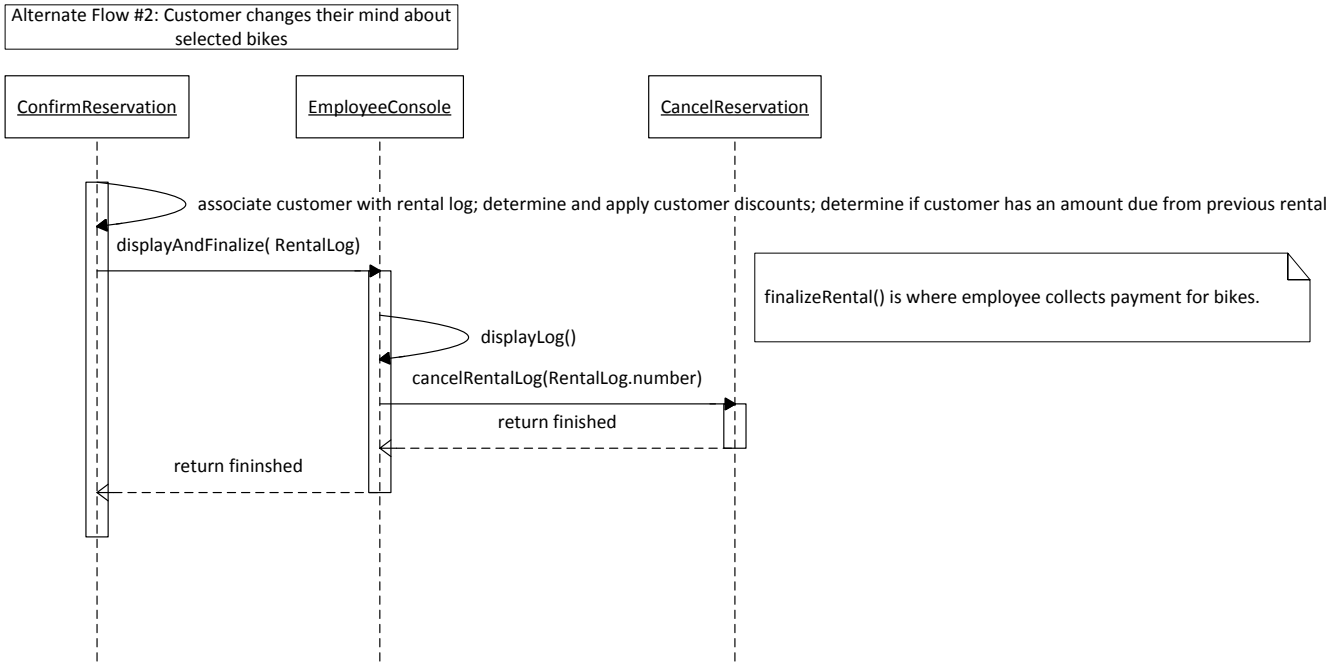




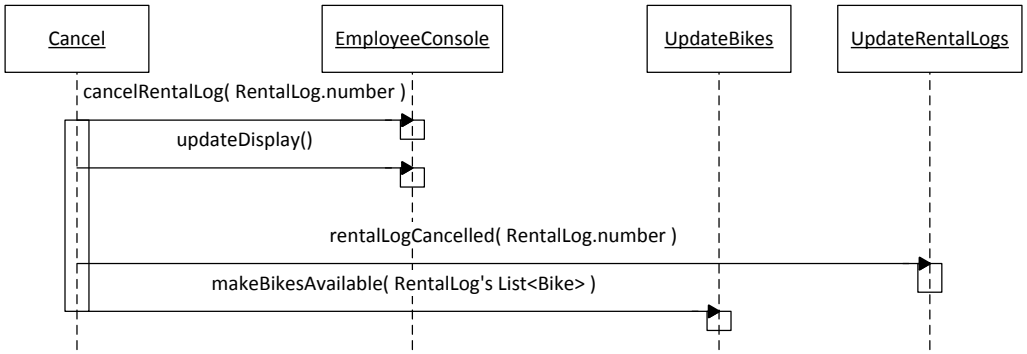
Confirm Reservation



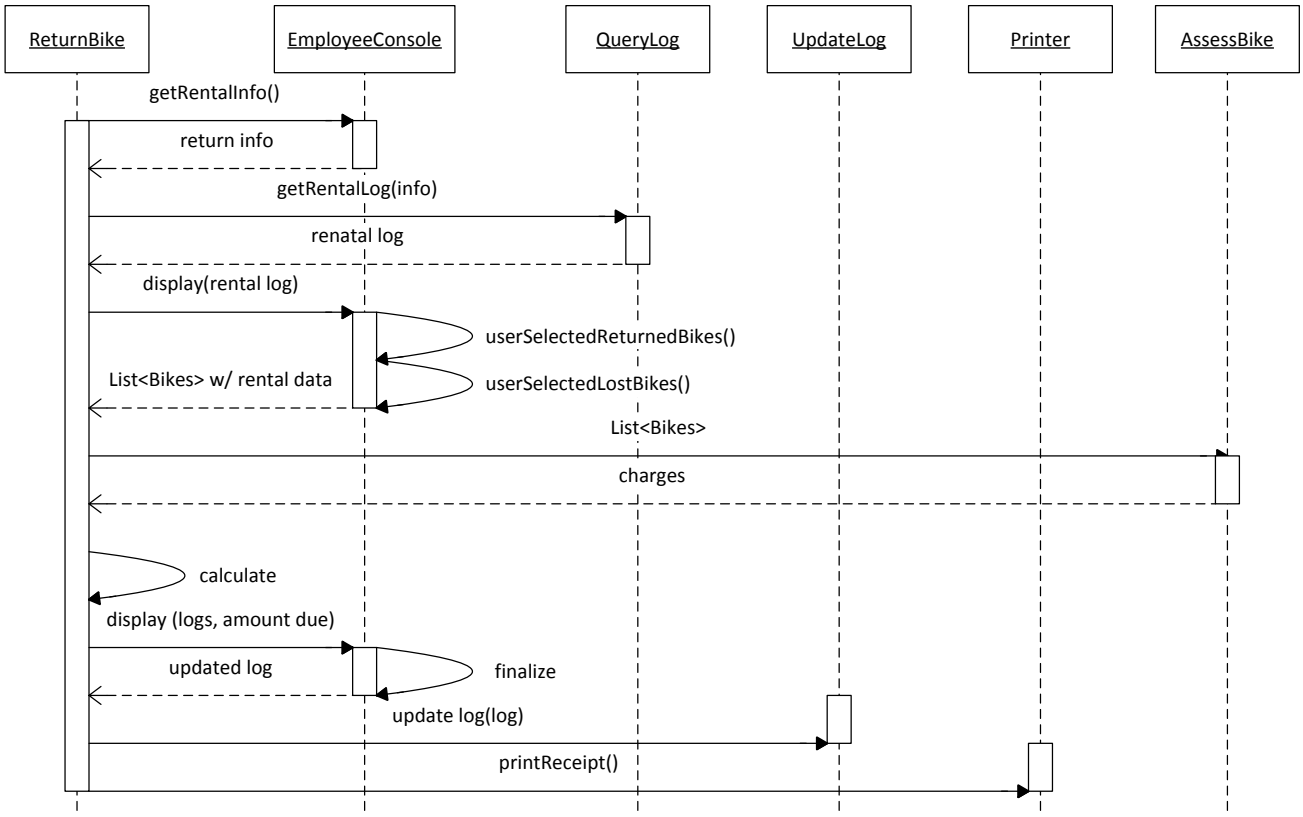

Alternate Flow #1: Customer cannot pay for rentals
ConfirmReservation
EmployeeConsole
CancelReservation
associate customer with rental log; determine and apply customer discounts; determine if customer has an amount due from previous rental
displayAndFinalize(RentalLog)
finalizeRental() is where employee collects payment for bikes.
displayLog()
cancelRentalLog(RentalLog.number)
return finished
return fininshed


Alternate Flow #2: Customer changes their mind about selected bikes
ConfirmReservation
EmployeeConsole
CancelReservation
associate customer with rental log; determine and apply customer discounts; determine if customer has an amount due from previous rental
displayAndFinalize(RentalLog)
finalizeRental() is where employee collects payment for bikes.
displayLog()
cancelRentalLog(RentalLog.number)
return finished
return fininshed

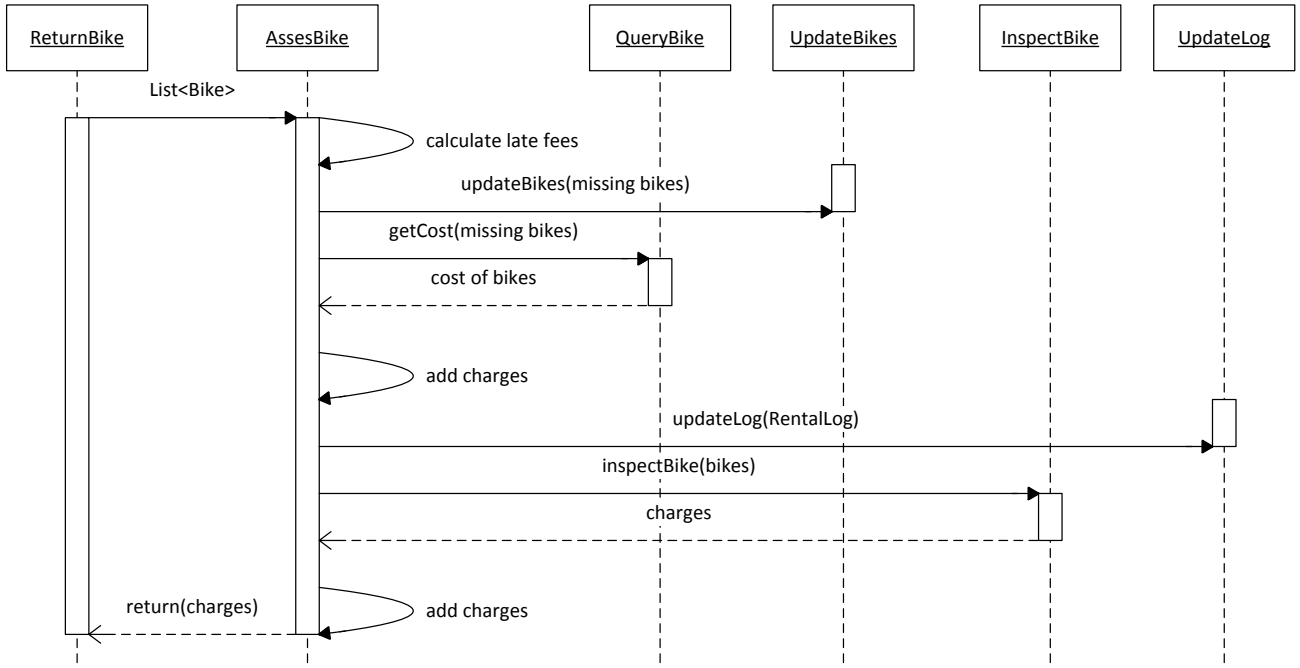
Cancel Reservation



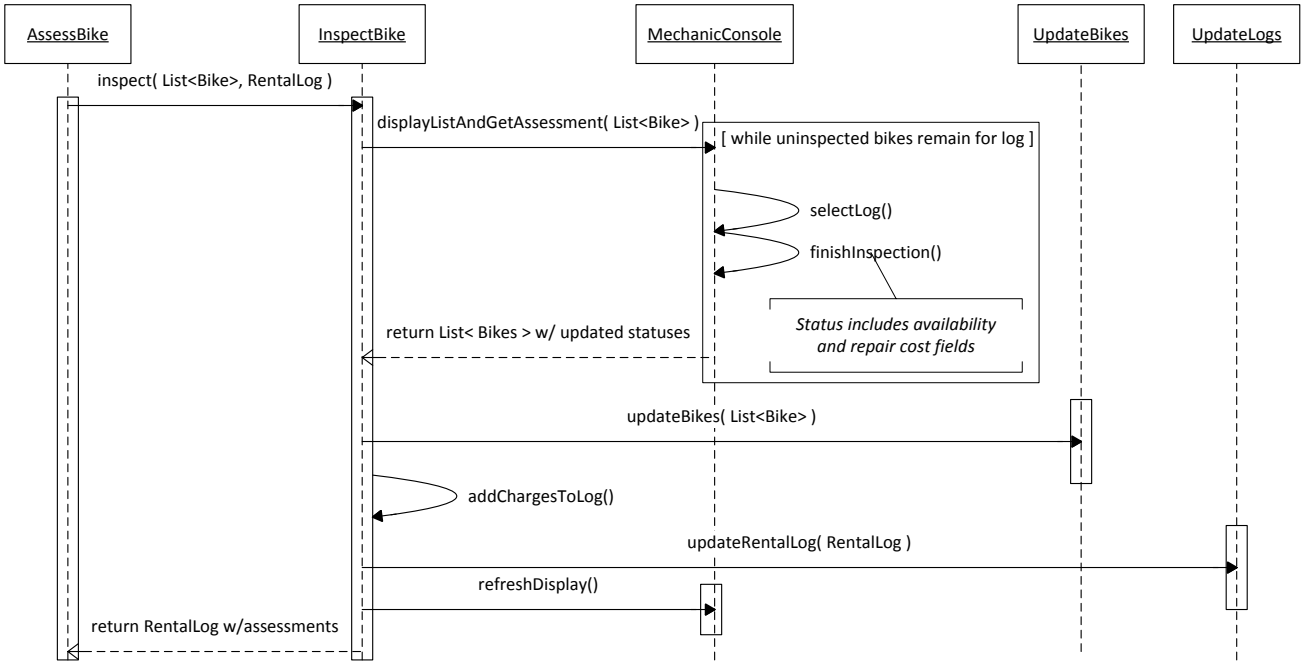
Return Bike



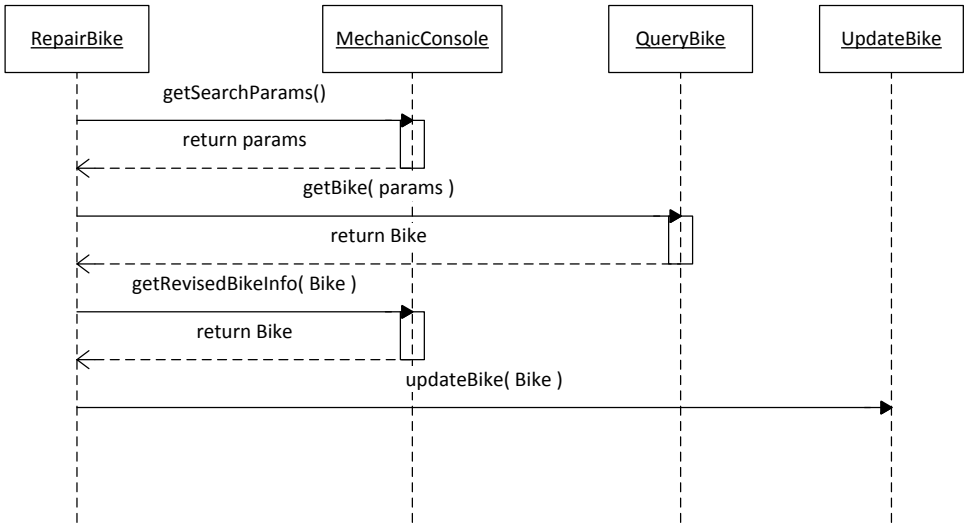
Assess Bike



Inspect Bike



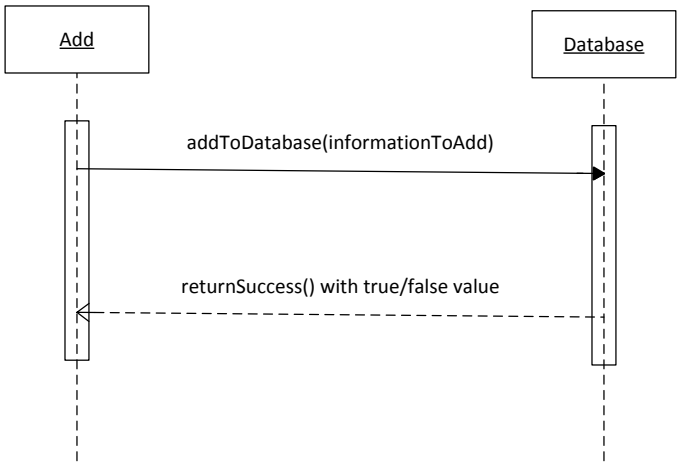
Repair Bike



Database Sequence Diagrams

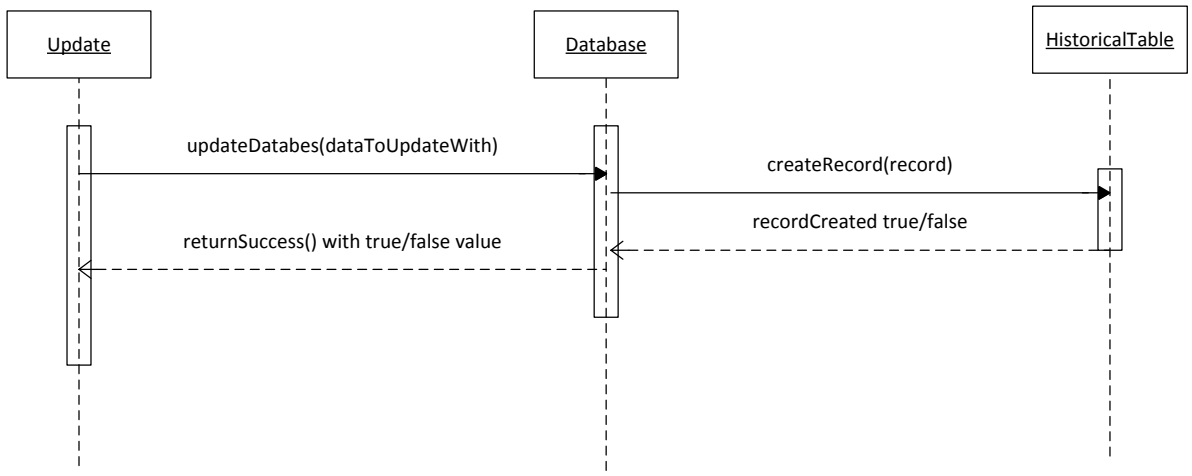
As database sequence diagrams rapidly become repetitive, as the same pattern is repeated, only a prototype pattern for each type of database transaction will be shown.

Add Record (Insert)



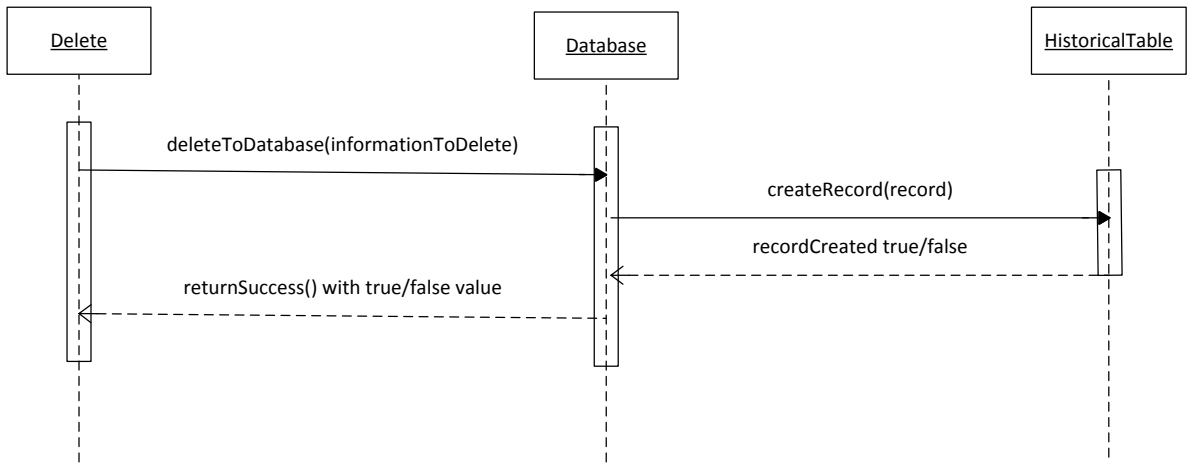
Sequence diagram of all add database type use case, use cases that creates a new record in the database

Update Record



Sequence diagram of all update database type use case. Diagram all database use cases that update a record in the database.

Delete Record



Sequence diagram of all delete database type use case, use cases that removes a new record in the database

INITIAL FUNCTIONAL TESTS

Notes: the Repair Bike Use Case, functionally, can be tested as a simple database transaction. (See UpdateBikes). Also, updated Functional Test information based on design phase changes is marked in red.

Use Case	Function Being Tested	Initial System State	Input	Expected Output
RentBike				
SelectBikes	Displaying previously selected bikes	Appropriate console ready to perform new bike search	Search Parameters	Appropriate console displays bikes selected on a previous search.
SelectBikes	Displaying available bikes	Appropriate console ready to perform new bike search	Search parameters	Appropriate console displays available bikes matching search parameters.
SelectBikes	Displaying available bikes	Appropriate console ready to perform new bike search	Category to browse	Appropriate console displays available bikes for selected category.
SelectBikes	Selecting bikes for rental	List of available bikes is displayed to operator.	Select bike(s) to rent.	Appropriate console displays the bikes with input field for rental_duration.
SelectBikes	Entering or editing rental dates for each selected bike	Operator has selected one or more bikes for rental.	- rental_duration	Form on appropriate console displays the updated information.
SelectBikes	Deselecting bikes	Operator has selected one or more bikes for rental. - OR - Form contains complete or partial rental information for bikes.	- select checkbox	Deselected bikes are removed from the form.
CreateRental Log	Bike Status changed to Reserved and Rental Log created	List of chosen bikes confirmed	- Database Transaction	Bike status changed to Reserved and Rental Log number returned
$SelectBikes_{cust}$ CreateRental Log	Confirming bike selection	Form contains complete rental information for bikes.	- finish selection button	Customer Console displays the rental log number.

Use Case	Function Being Tested	Initial System State	Input	Expected Output
SelectBikes$_{cust}$	Confirming rental log number	Customer Console displaying the rental log number	- acknowledge button	Customer Console displays message to see available employee Rental log number for corresponding bikes is displayed on Employee Console as a hyperlink whose action is to identify the customer.
SelectBikes$_{emp}$ CreateRental Log	Confirming bike selection	Form contains complete rental information for bikes.	- finish selection button	The rental log number is displayed on the Employee Console.
SelectBikes Cancel Reservation	Cancelling bike selection	Form contains complete information.	- cancel selection button	Appropriate console indicates rental is cancelled.
SelectRentalLog	Select rental log number to process next	Employee Console lists rental log numbers that are ready for service	- employee selects log number	The rental log information is displayed on Employee Console in a state where it is ready to be updated with customer information. The selected log should be removed from all other Employee Consoles.
Select Customer	Identifying an existing customer	Customer search screen is displayed on the Employee Console	Search parameters	A list of customer matches is displayed on the Employee Console, with the ability to select one.
Select Customer	Selecting an existing customer	The searched list of customers is displayed on the Employee Console	- select customer button	The selected Customer is chosen and displayed.
Select Customer	Adding a new customer	Customer not found by search - OR - Customer known to be new customer	Customer data fields	A new customer record is created.
Select Customer Confirm Reservation	Verifying frequent customer discounts	Rental log is complete with bike information and customer information	Customer and their frequency	The correct discount should be applied to the bill displayed.
Select Customer Confirm Reservation	Connecting a customer to a rental log and recording payment.	Rental log is complete with bike information and customer information	- payment - confirm button	Employee Console indicates that the reservation was successfully committed to the database.

Use Case	Function Being Tested	Initial System State	Input	Expected Output
Confirm Reservation	Print a receipt for rental	Rental has been paid for and finalized by employee	Employee has finalized the rental	The bike statuses should be updated to rented. A receipt of the transaction is printed. The employee console is reset to its initial state.
Cancel Reservation	Reserved Bikes set back to available and Rental Log set to cancelled	Rental Log is created and Bikes are reserved	- Cancel button	The bikes statuses are set back to available and the Rental Log is set to cancel
Return Bike				
Return Bike	Entering rental info	Employee console ready for rental log search	Enter search parameters	List of bike(s) rented by the customer is displayed on the console
Return Bike Assess Bike Inspect Bike	Marking the status of returned and lost bikes	Employee console displays the full list of bike(s) rented by the customer	Choose which Bikes are being returned and which Bikes are lost	The returned bikes and lost bikes are displayed on the Employee Console with the appropriate late fees, repair fees and replacement fees.
Return Bike	Calculate additional customer charges / refunds based on the deposit.	All charges have been calculated or entered by Asses Bike / Inspect Bike	none	The rental log displays the final moneys due to the customer or due to the bike shop.
Return Bike	Rental log contains updated information	All charges have been calculated or entered by Asses Bike / Inspect Bike	Employee finalizes the transaction	The Employee console verifies that the rental log was updated.
Return Bike	Print a receipt	All charges have been calculated or entered by Asses Bike / Inspect Bike	Employee finalizes the transaction	Receipt is printed correctly. The employee console is reset to its initial state.
Assess Bike	Update status of lost bikes	List of lost bikes has been entered into the system	lost bikes	Status of bikes in database is changed to indicate the bike is lost.
Inspect Bike	Processing an inspection request.	The list of *previous* rental logs for returned bikes that have not yet been inspected is displayed.	Select a rental log to process.	System displays the returned bikes which correspond to selected rental log with input fields for repair_amount and bike_availability.

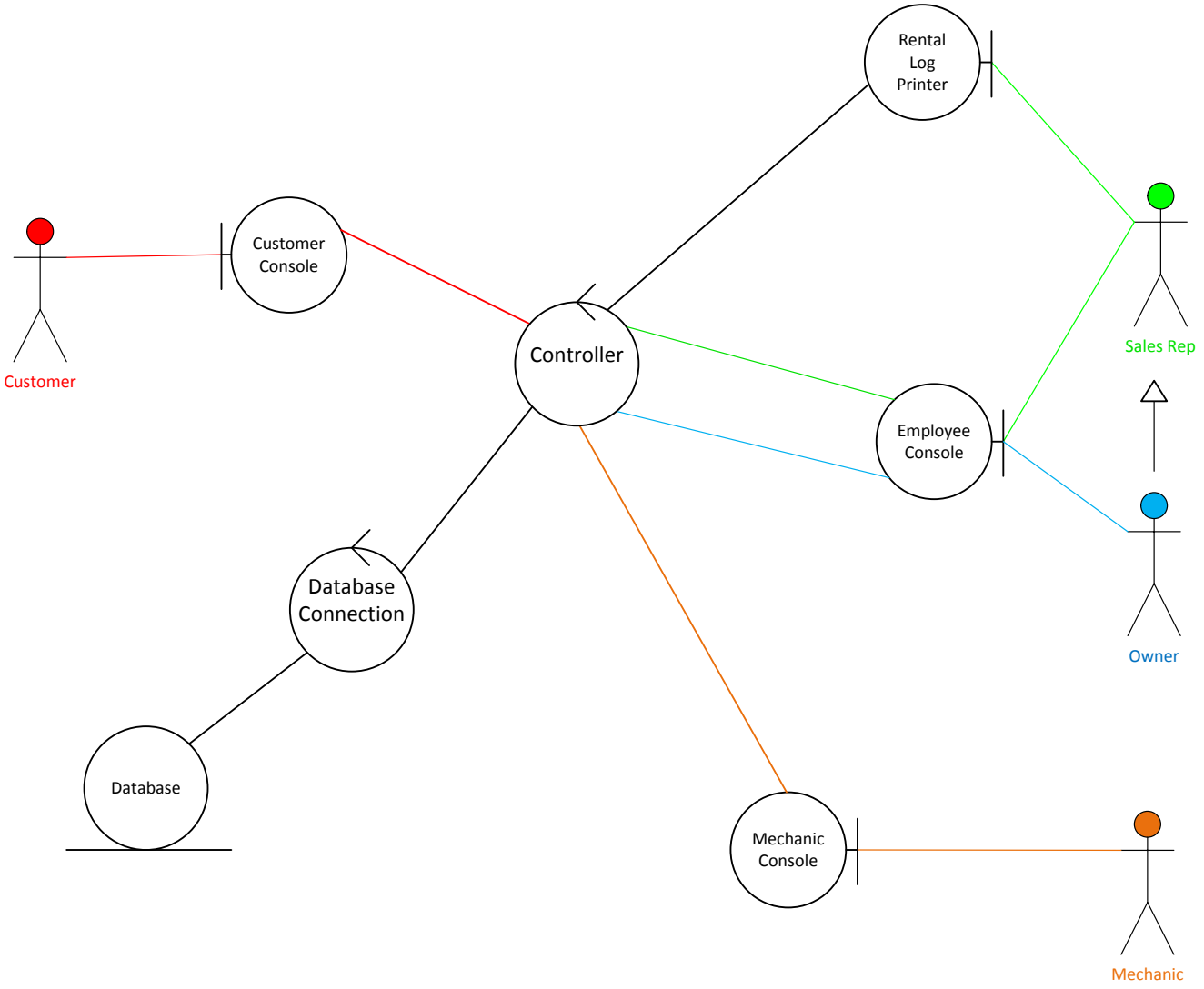
Use Case	Function Being Tested	Initial System State	Input	Expected Output
Inspect Bike	Updating bike information	Bike record(s) with input fields for repair_amount and bike_availability are displayed.	- repair _amount - bike _availability - inspected checkbox	Form displays updated information. Inspected bikes display the current date as the inspection_date.
Inspect Bike	Finalizing an inspection for a rental log	The selected rental log indicates on the display that all returned bikes have been inspected for this rental log	- finish inspection button	System indicates successful completion of database transactions and displays the list of any rental logs for returned bikes that have not yet been inspected. This list no longer includes the rental log that was just finished. The mechanic console is reset to its initial state.
Inspect Bike	Finalizing an inspection for a rental log	The selected rental log indicates on the display that **not** all returned bikes have been inspected for this rental log.	- finish inspection button	System prompts user to finish the inspecting all returned bikes for the rental log.
Repair Bike	Entering search parameters	A list of bike attributes and corresponding input boxes is displayed.	- searchable bike attributes	System displays a list of bikes matching the search criteria
Repair Bike	Selecting bike repaired	List of bikes matching search criteria is displayed on the mechanic console	- mechanic selects bike	System displays the bike with input fields
Repair Bike	Updating bike information.	Bike record(s) with input fields for repair_amount and bike_availability are displayed.	- bike _availability - repaired checkbox	Form displays updated information. Repair_amount is set to 0. Bikes display the inspection_date as the current date. Bike becomes available
Repair Bike	Recording bike repairs.	Each bike record(s) displays the updated status.	- record repairs button	System indicates successful completion of database transactions.
Database Transactions				
Add Customer	Insert record	Input screen for customer information	- customer data - commit button	Database contains the inserted row.
Update Customers	Update record	Input screen for chosen customer information	-customer chosen -commit button -customer data	Database contains the updated information in chosen customer row.
Query Customers	Search records	Input screen for customer attributes	-customer attributes -search button	Screen contains a list of customers to choose from
Remove Customers	Delete record	Input screen to choose customer	-customer chosen -delete button	Database moves the selected row, to inactive customers.

Use Case	Function Being Tested	Initial System State	Input	Expected Output
Acquire Bikes	Insert record	Input screen for bike information	-bike data -commit button	Database contains the inserted row
Update Bikes	Update record	Input screen for chosen bike information	-bike chosen -commit button -bike data	Database contains the updated information in chosen bike's row.
Query Bikes	Search records	Input screen for bike attributes	-bike attributes -search button	Screen contains a list of bikes to choose from
Decommission Bikes	Delete record	Input screen to choose bike	-bike chosen -delete button	Database removes the selected row.
Add Logs	Insert record	Rent action	-bikes chosen -detail information	Database contains the inserted row
Update Logs	update record	Rent or Return case	-log information -detail information	Database contains the updated row.
Query Logs	search record	Rent or Return case	-search attribute	List of logs to choose from
Add Details	insert record	Add Log or Update Logs	-chosen log -detail information	Log has inserted detail
Update Details	update record	Update Logs	-chosen log -chosen detail information	Log has updated detail information
Query Details	search record	Update Logs or Query Logs	-chosen log -search attribute	List of details to choose from
Delete Details	delete record	Rent or Return case	-detail information	Log has detail deleted
Logging In				
Logging In	customer login	Customer Console displays a login button	None, customer presses button	"Login" successfully processes and the list of controllers appropriate to the Customer console (i.e., SelectBikeCustomer) displays as buttons.
Logging In	employee login	Employee Console has input prompts for credentials.	Incorrect credentials -username -password	Login is denied

Use Case	Function Being Tested	Initial System State	Input	Expected Output
Logging In	employee login	Employee Console has input prompts for credentials.	Correct Sales Rep credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Customer console and Sales Rep (e.g., RentBikeEmployee, ReturnBike) displays as buttons. Each rental log of state InRentalLog is displayed as a selectable item
Logging In	employee login	Employee Console has input prompts for credentials.	Correct Owner credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Customer console and Owner (e.g., ReturnBike, DecommissionBike) displays as buttons. Each rental log of state InRentalLog is displayed as a selectable item
Logging In	employee login	Employee Console has input prompts for credentials.	Correct Mechanic credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Customer console and Mechanic (i.e., **nothing**!) displays as buttons. Each rental log of state InRentalLog is displayed as an **unselectable** item
Logging In	employee login	Mechanic Console has input prompts for credentials.	Incorrect credentials -username -password	Login is denied
Logging In	employee login	Mechanic Console has input prompts for credentials.	Correct Sales Rep credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Mechanic console and Sales Rep (i.e., **nothing**!) displays as buttons. Each rental log of state BeingInspected is displayed as an **unselectable** item
Logging In	employee login	Mechanic Console has input prompts for credentials.	Correct Owner credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Customer console and Owner (i.e., **nothing**!) displays as buttons. Each rental log of state BeingInspected is displayed as an **unselectable** item
Logging In	employee login	Mechanic Console has input prompts for credentials.	Correct Mechanic credentials -username -password	"Login" successfully processes and the list of controllers appropriate to the Mechanic console and Mechanic (e.g., Repair Bike) displays as buttons. Each rental log of state BeingInspected is displayed as a selectable item

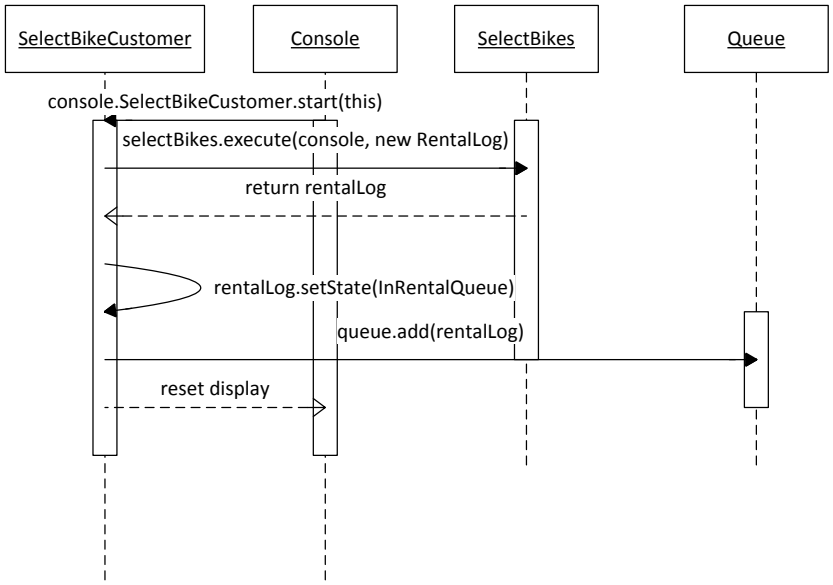
Design Documents

REVISED ANALYSIS CLASSES

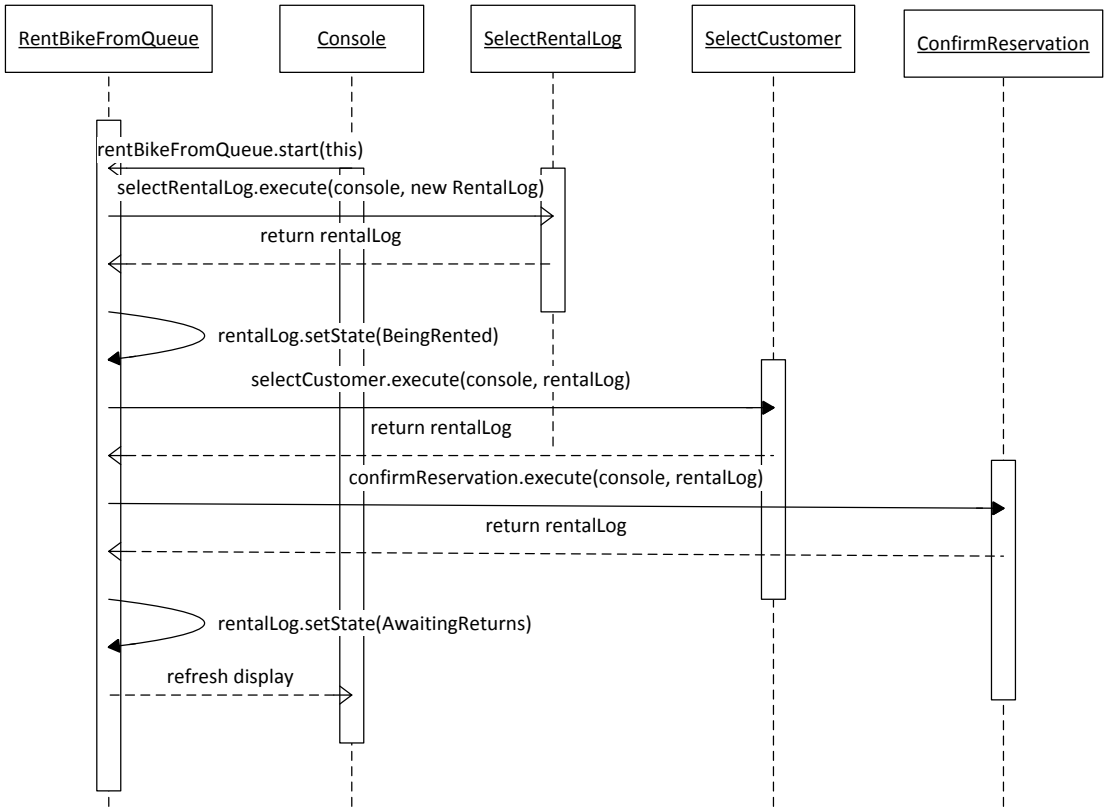


REVISED SEQUENCE DIAGRAMS

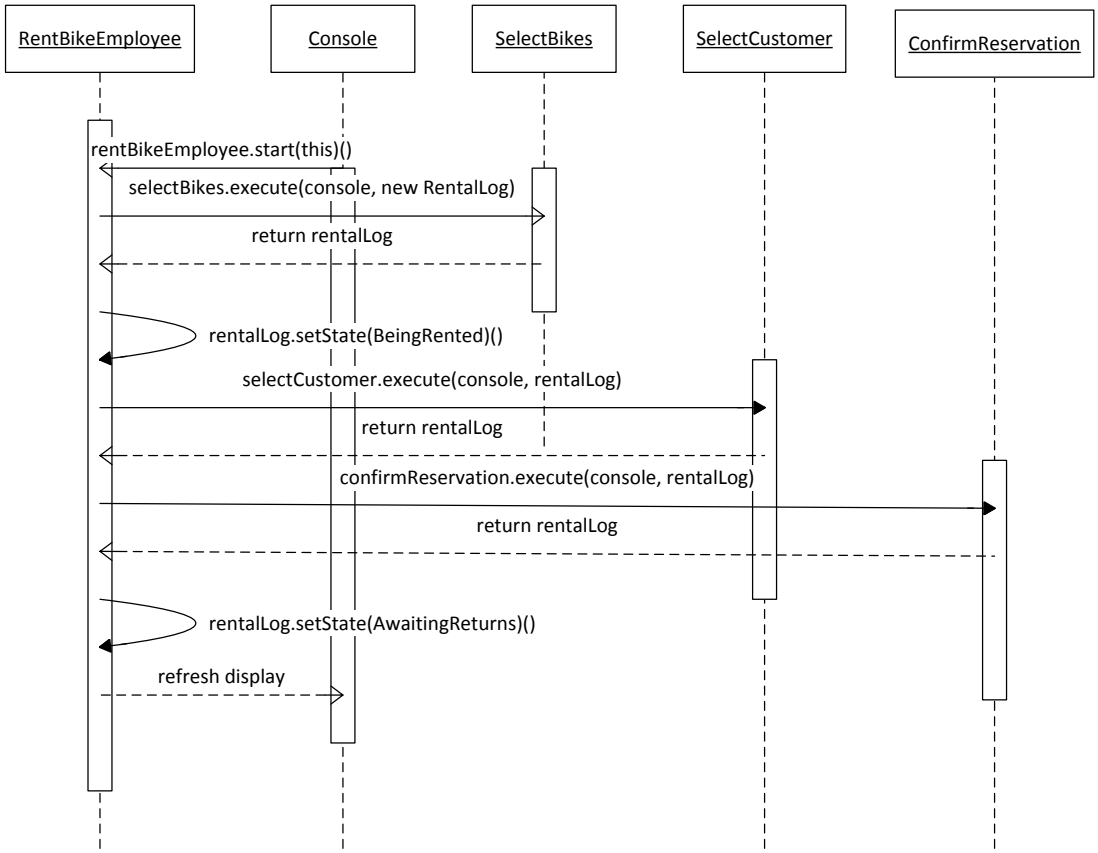
Select Bike Customer



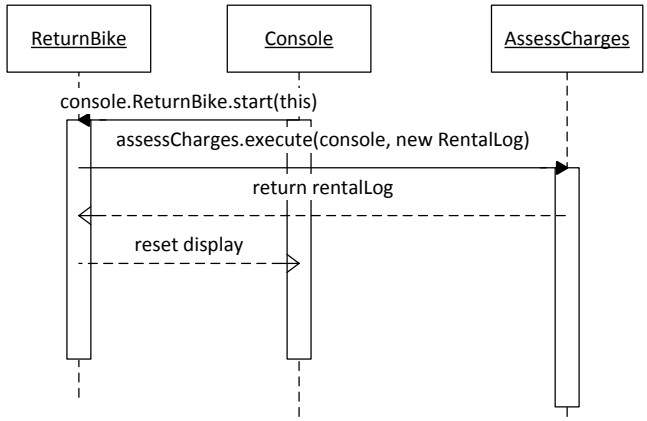
Rent Bike From Queue



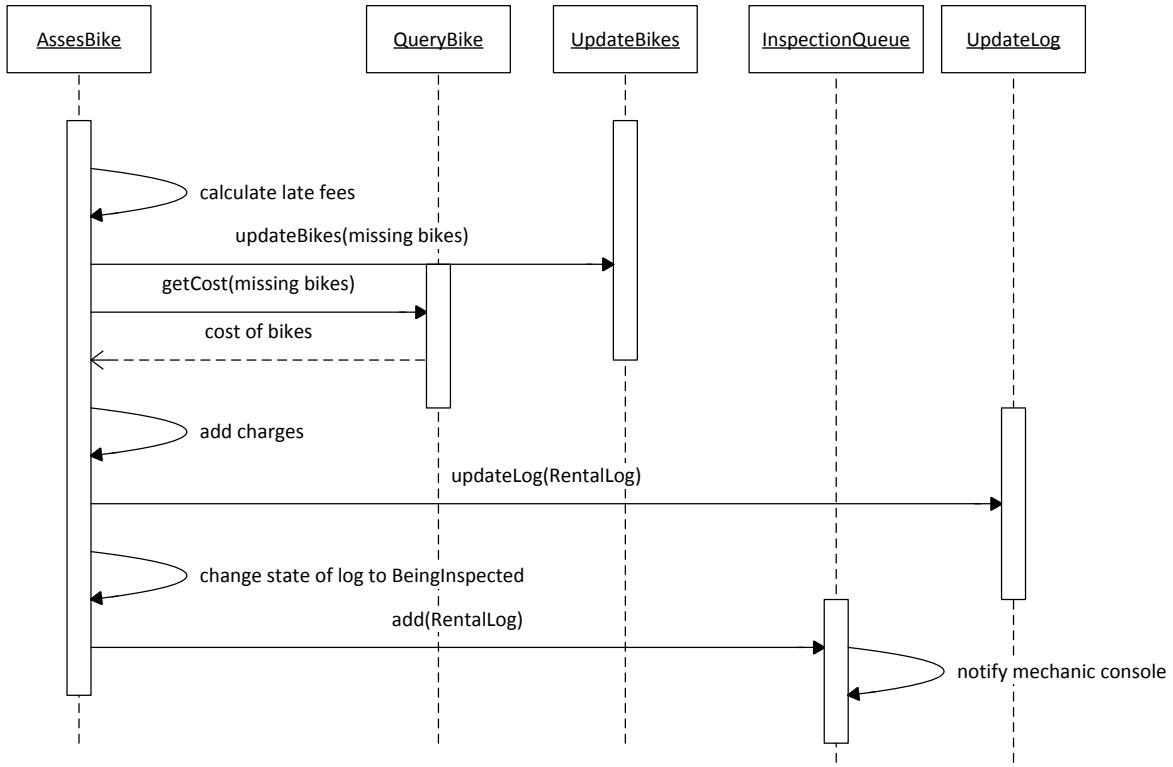
Rent Bike Employee



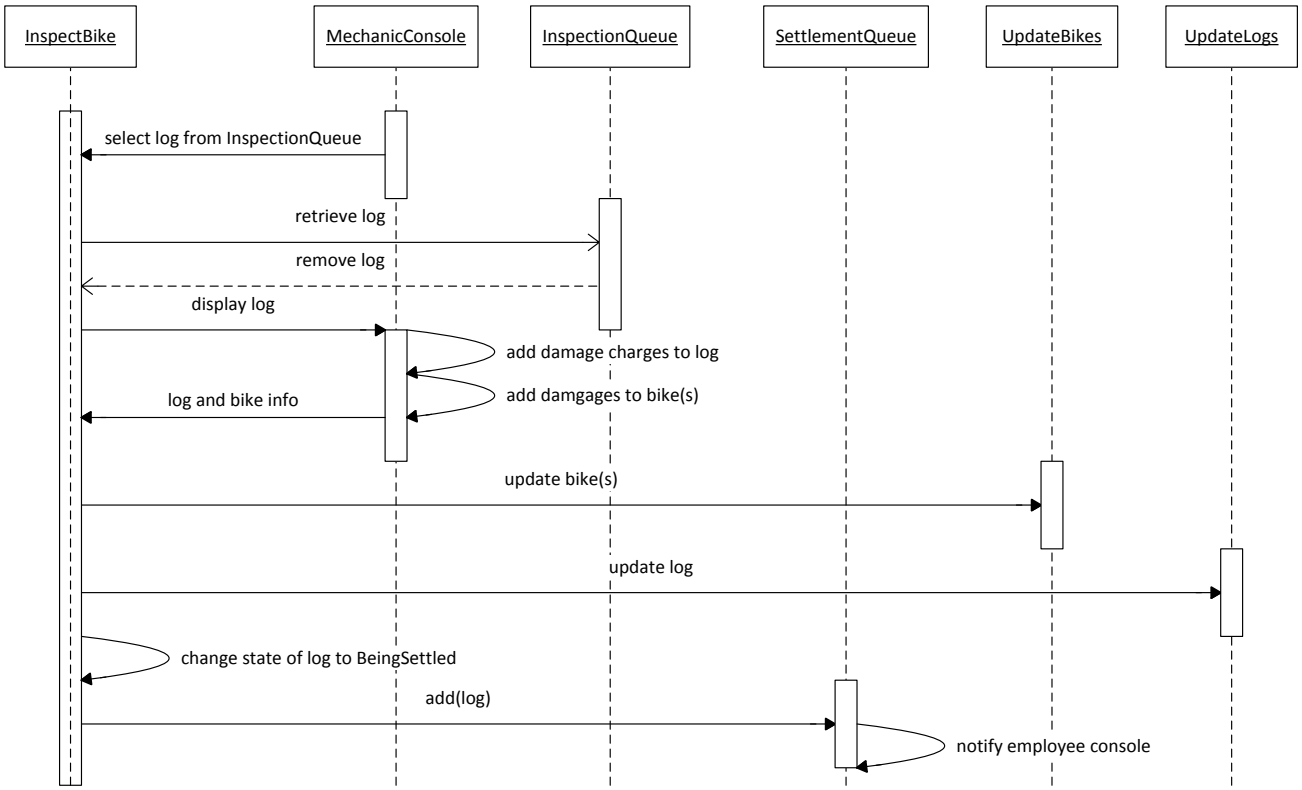
Return Bike



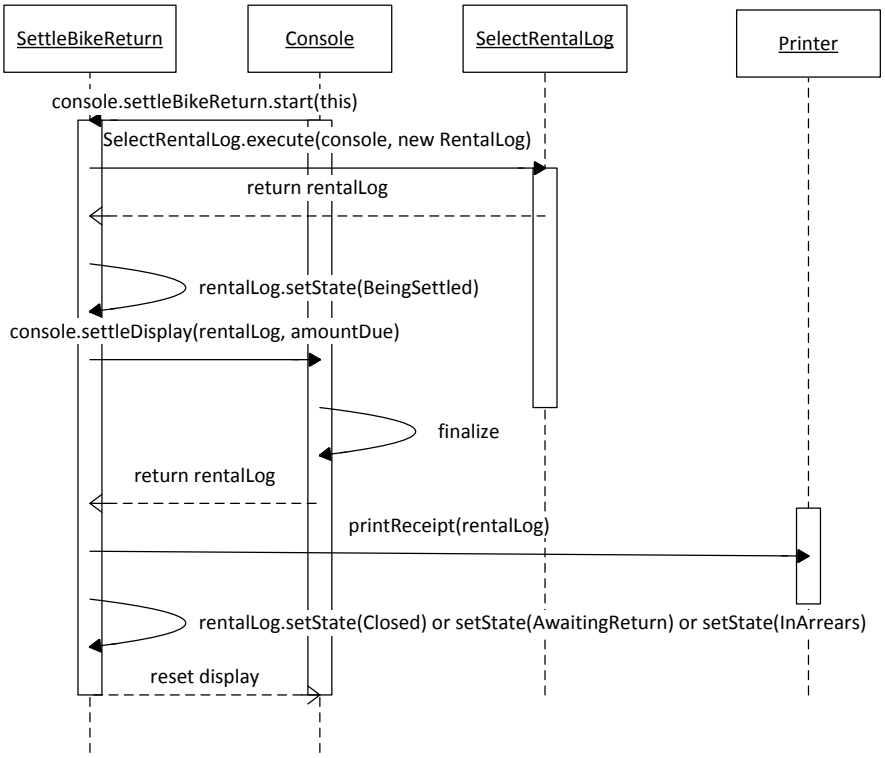
Assess Charges



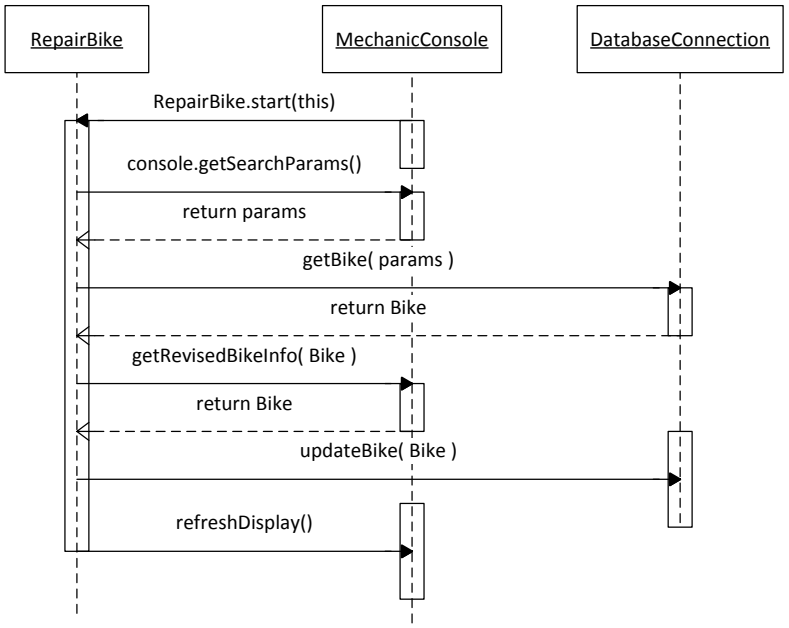
Inspect Bike



Settle Bike Return



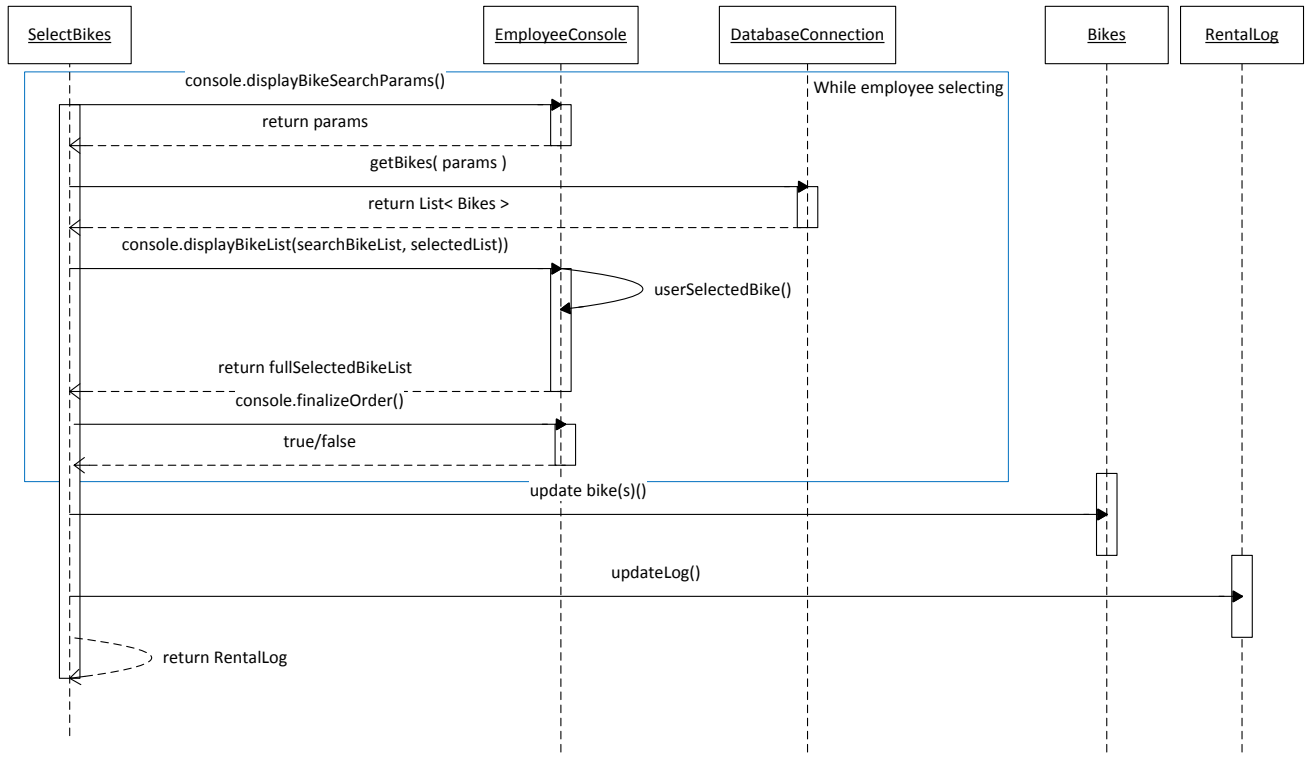
Repair Bike



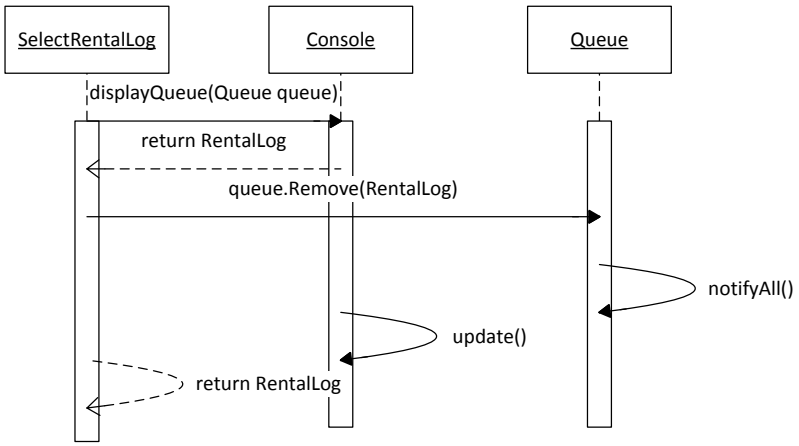
Reused Subcontrollers

These subcontrollers are utilized by multiple controllers.

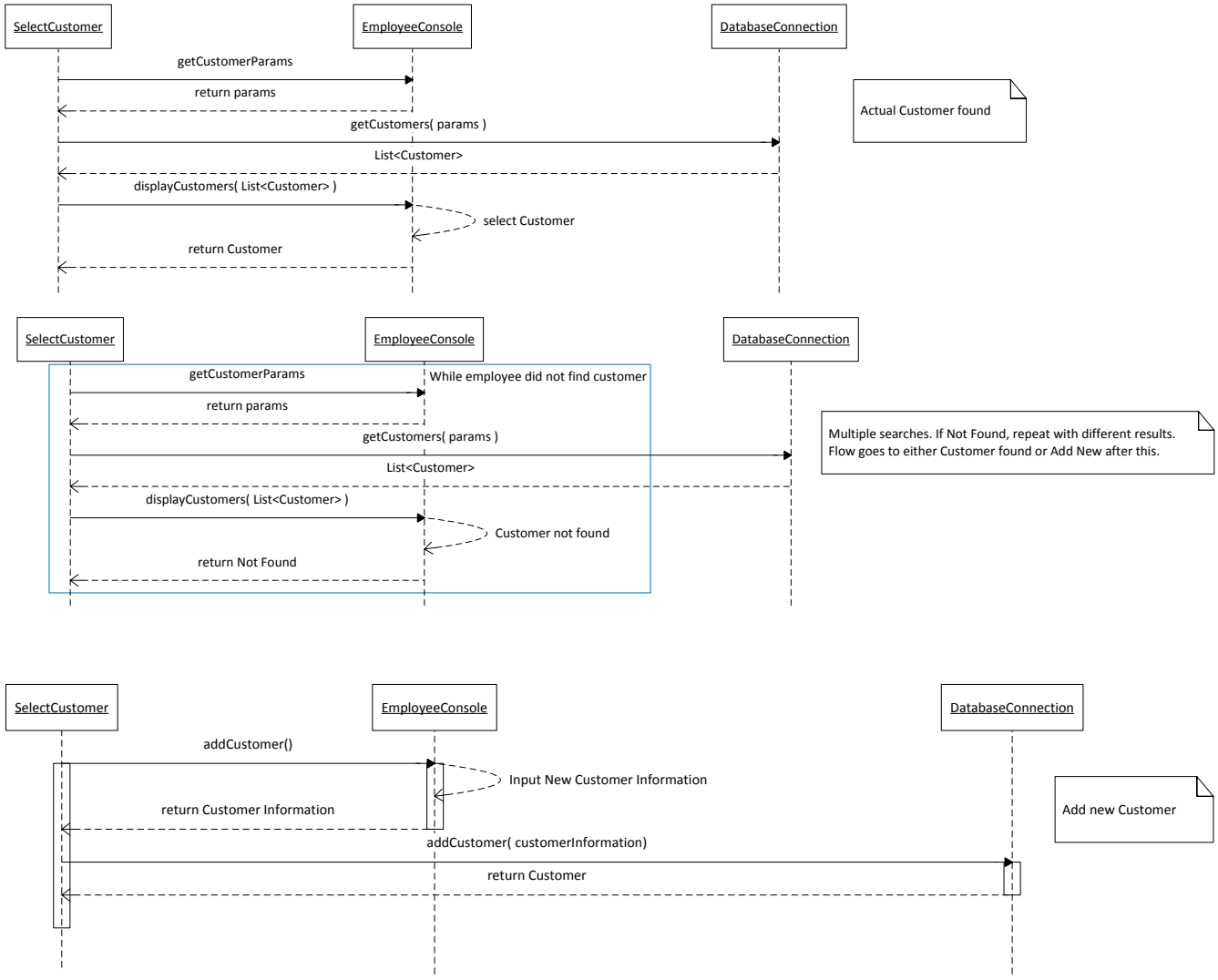
Select Bikes



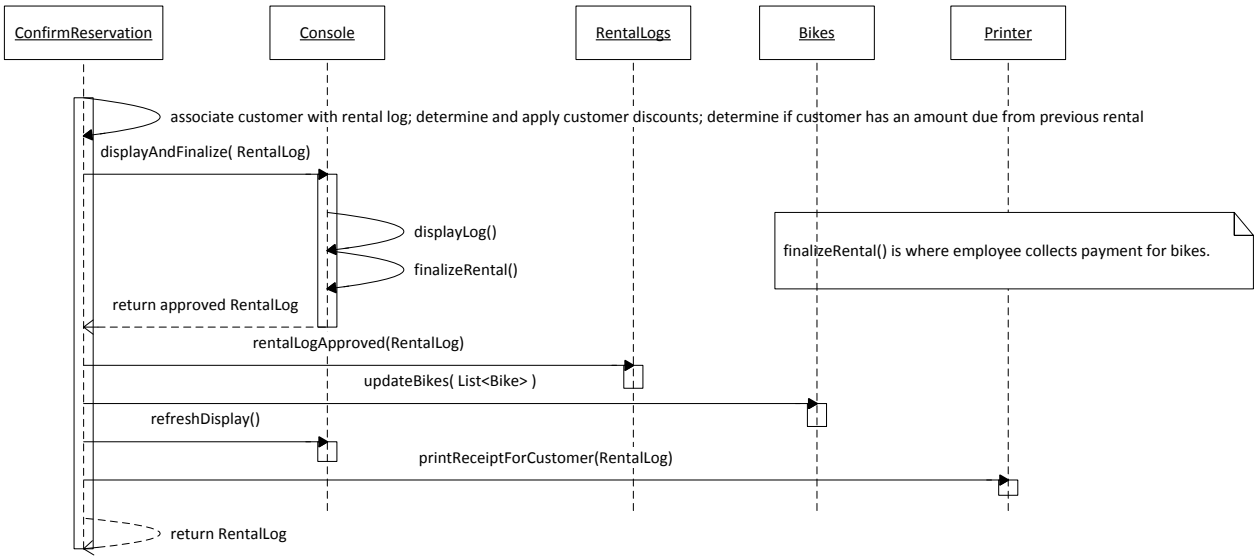
Select Rental Log

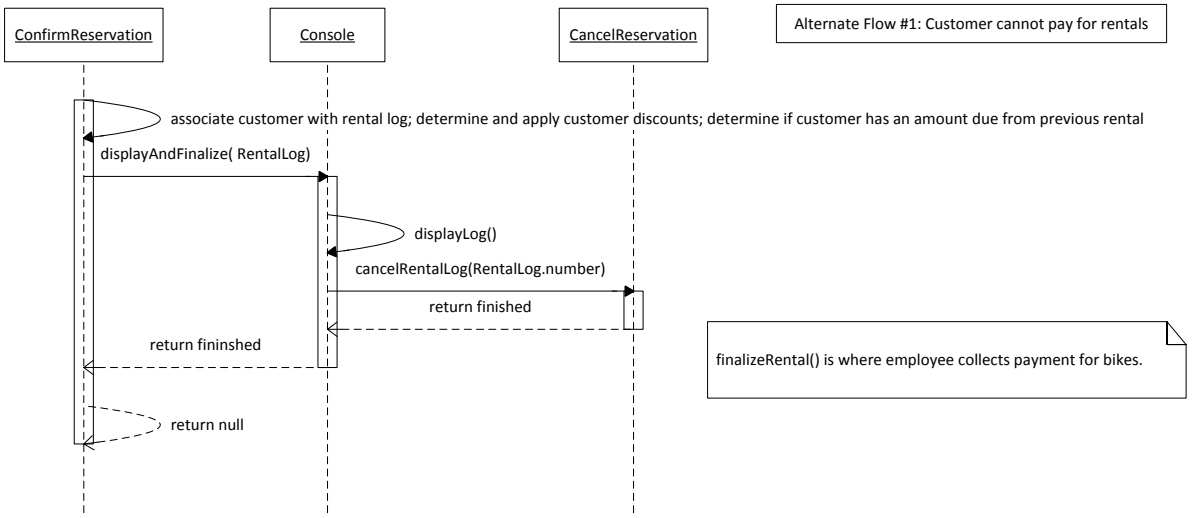


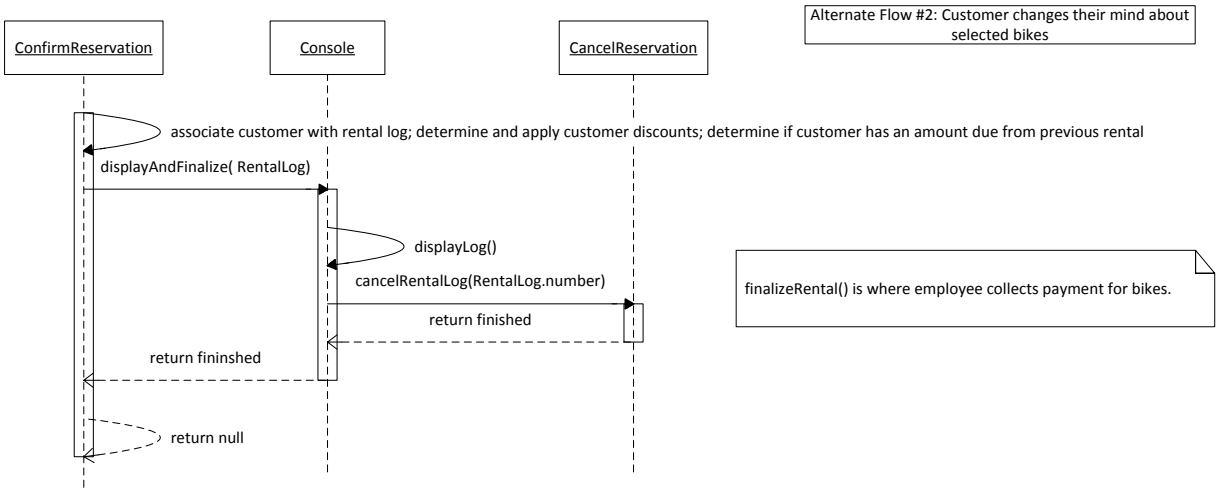
Select Customer



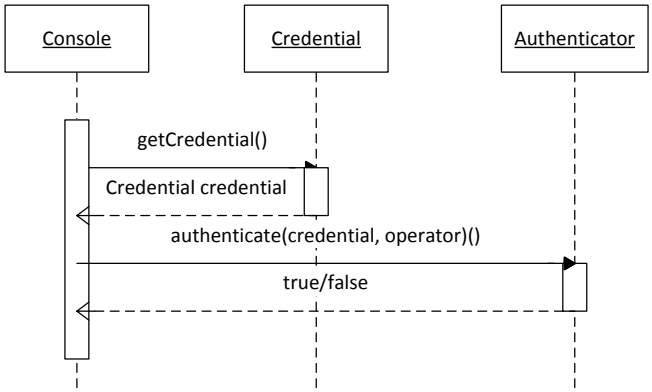
Confirm Reservation





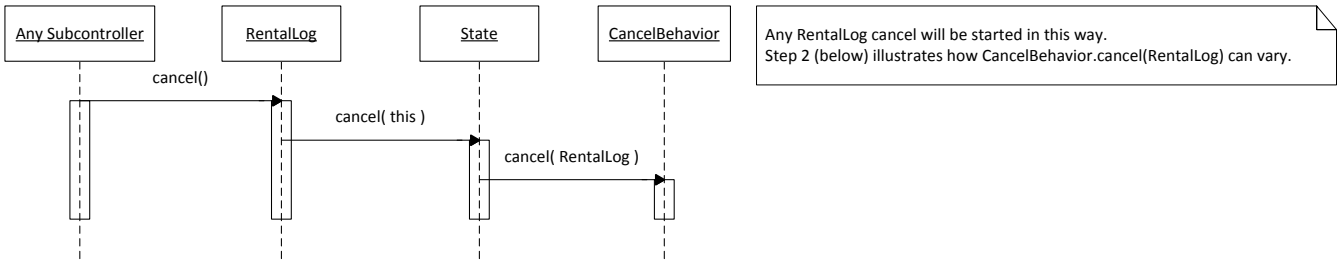


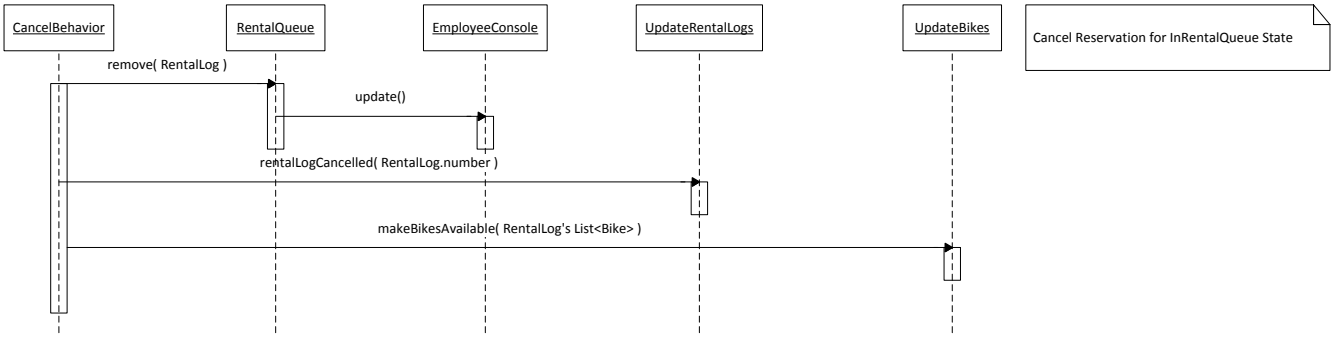
Login

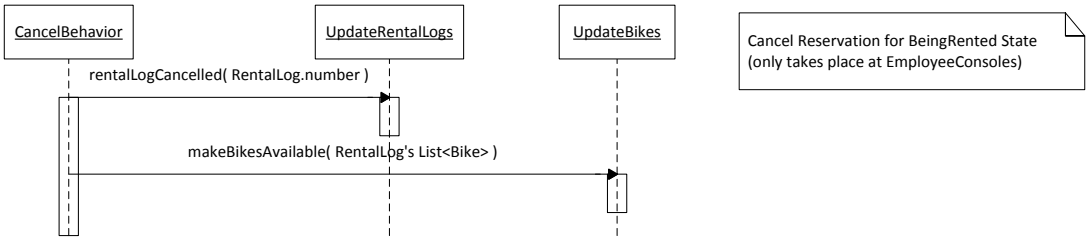


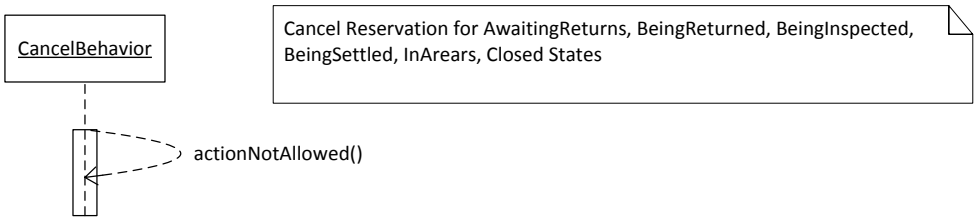
Cancel Reservation

Step 1: Cancels are started from subcontroller



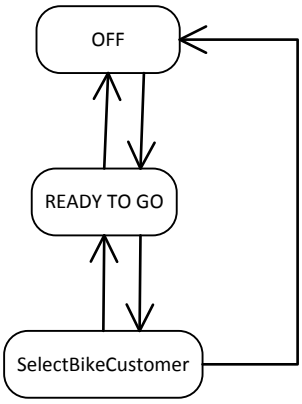




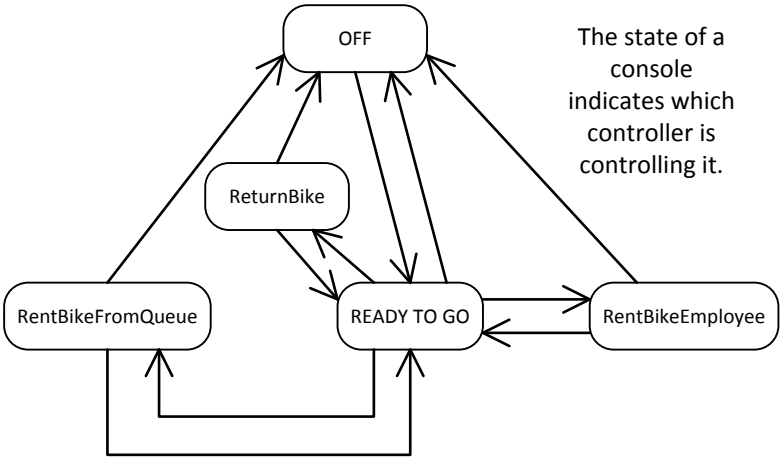

CancelBehavior
Cancel Reservation for AwaitingReturns, BeingReturned, BeingInspected, BeingSettled, InArears, Closed States
actionNotAllowed()

STATE DIAGRAMS

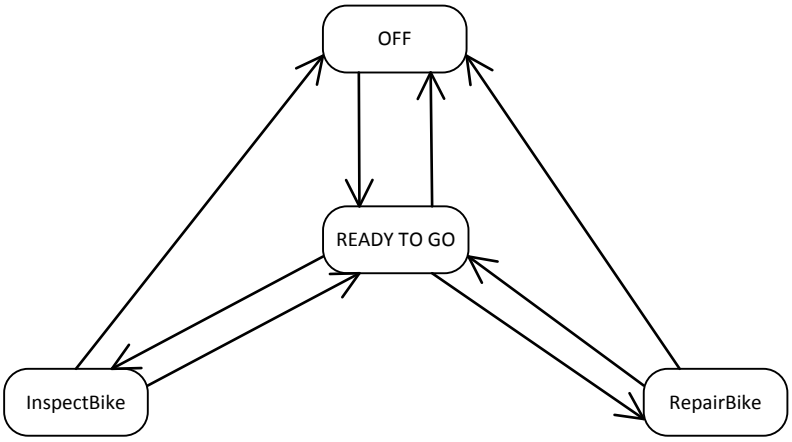
Customer Console



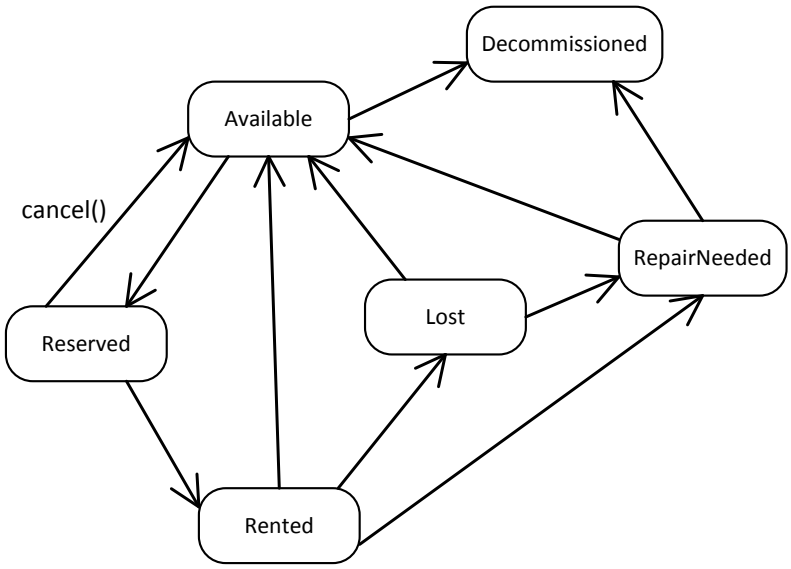
Employee Console



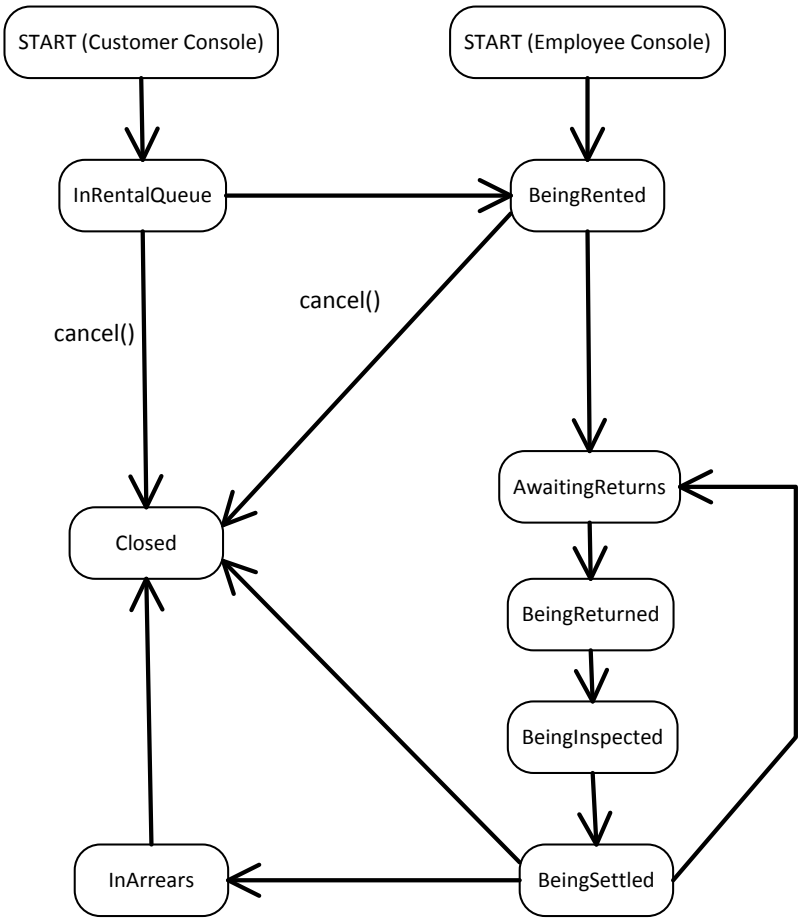
Mechanic Console



Rentables



Rental Logs



DESIGN SUMMARY

Database Operations

Database operations are handled by their own set of controlling classes (identified by the yellow ovals on the Use Case Diagram). This approach insulated all database actions from the business logic of the Bike Shop. Each database operation would employ a DatabaseConnection object via the **Singleton** pattern. This pattern controls the creation of a database connection and (in the current design state) keeps delivering the same connection to each database class (e.g., Add Bike) that needs it. The advantages of using **Singleton** here is that we don't allow every class to create its own unique, resource-intensive database connections and it ensures database consistency for multiple transactions. This pattern is also extendable. If the Bike Shop ever expands to the scale where it needs database pooling, the getConnectionInstance() method could be modified to manage a connection pool and encapsulate the distribution of database connections.

Interaction Between Controllers, Subcontrollers and Consoles

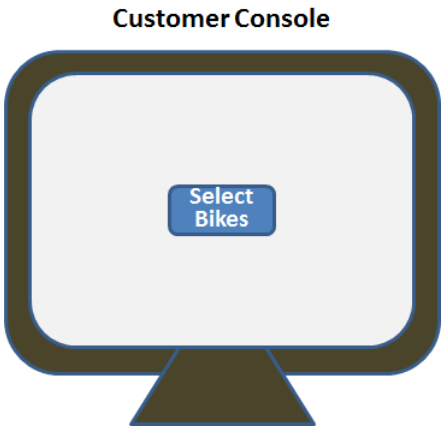
The Bike Shop application is designed using the **MVC** pattern at its core. Some operations were simple enough to be modelled with a single controller, for instance RepairBike (which is a basic function) and decommissioning a bike (which is a simple database operation). Other operations, specifically Rent Bike and Return Bike, were modelled in the Use Case diagram in a manner to suggest subcontrollers.

For example, RentBike was modeled in the *analysis phase* with the following subcontrollers:

Subcontroller	**Purpose**	**Why it was originally modelled as a subcontroller**
SelectBikes	To allow for iterative searching of the bike inventory and the selection of bikes	This is the only RentBike component that is visible to the customer.
CreateRentalLog	To build the rental log with selected bikes, and update the Bike and RentalLog database tables	Not directly exposed to the customer. Actually used by SelectBikes
SelectRentalLog	To allow employees to select a rental log created by a customer	This actually begins the *employee side* of customer-initiated RentBike operations
SelectCustomer	To allow a log to be augmented with customer data, including their discount	Encapsulated complex logic into its own subcontroller
ConfirmReservation	To finalize the rental	It was the last stage of RentBike
CancelReservation	To cancel a reservation prior to finalizing it.	To handle alternate flows, where a reservation is cancelled.

However, when moving into the *design phase*, it became more efficient to modify this design based on the interaction between consoles and controllers/subcontrollers.

Impact of Consoles on Controllers



When designing the consoles, we wanted to set them up so that they would have a button to begin a certain process. We also did not want the controllers to have any business logic, to adhere to the **MVC** pattern.

The abstract class Console will maintain a list of abstract Controllers. Concrete implementations of Console will be instantiated with a concrete controller. The Console will call the Controller's start() method (utilizing the **Command** pattern). From that point onward, the Controller will direct the display of the Console and receive its output. The Console, then, is largely limited to being only a user interface.

This led us to reconsider which behaviors were governed by Controllers versus Subcontrollers. Essentially any operation that is initiated from any Console was modelled as a Controller.

Operation	**Analysis Phase Controllers and Subcontrollers**	**Design Phase Controllers and Subcontrollers**
Renting a bike	**RentBike** • SelectBikes ○ CreateRentalLog • SelectRentalLog • SelectCustomer • ConfirmReservation • CancelReservation	**SelectBikeCustomer** • SelectBikes **RentBikeFromQueue** • SelectRentalLog • SelectCustomer • ConfirmReservation **RentBikeEmployee** • SelectBikes • SelectCustomer • ConfirmReservation
Returning a bike	**ReturnBike** • AssessCharges ○ InspectBike	**ReturnBike** • AssessChages **InspectBike** • SelectRentalLog **SettleBikeReturn** • SelectRentalLog
Repair bike	**RepairBike**	**RepairBike**

This also led to the creation of Queues which will be discussed in a further section.

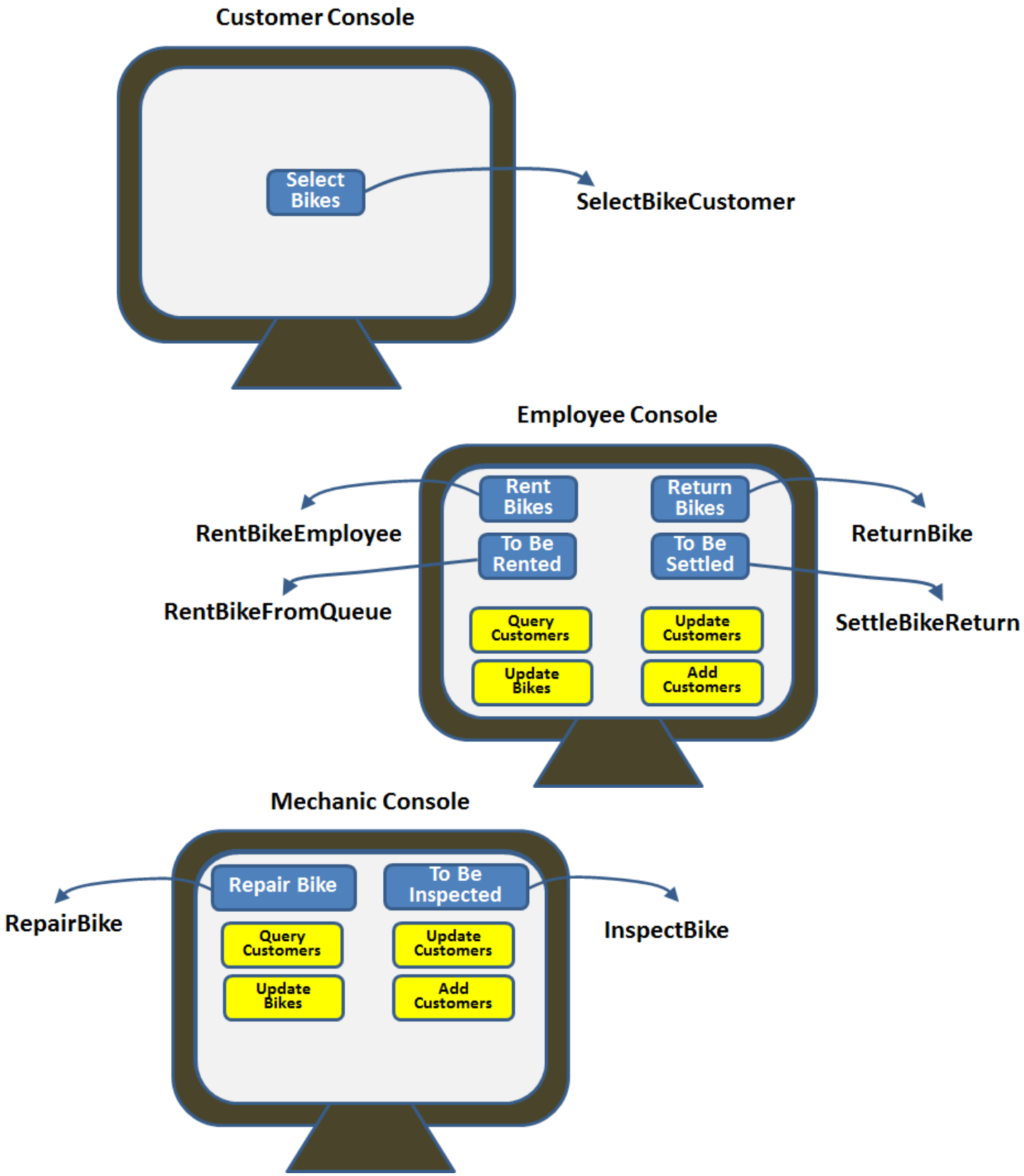
Notes:

- SelectBikes, SelectCustomer,ConfimReservation and SelectRentalLog turned out to be very good subcontrollers as each was used by more than one Controller
- InspectBike was promoted to Controller, because of its association with the Mechanic Console. Likewise new controller SelectBikeCustomer was created because of its

association with the Customer Console, and RentBikeFromQueue and SettleBikeReturn were created because of their association with the Employee Console.

- CreateRentalLog was incorporated into the SelectBikes controller. (It was always suspect that a Subcontroller would have an additional Subcontroller, as posited in the analysis phase.)

Console / Controller Relationships



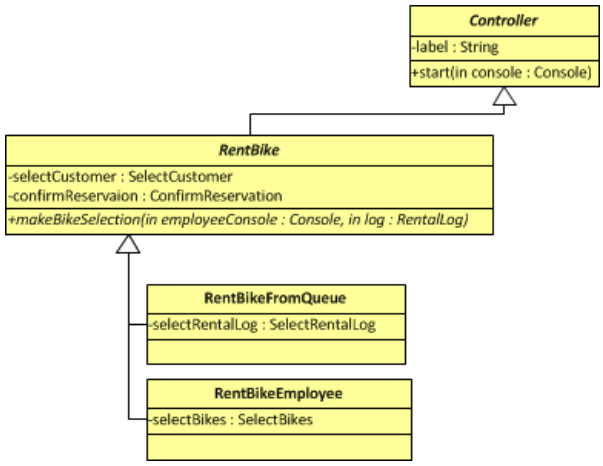
Note that the database operations (yellow) are themselves Controllers and are also available on the employee and mechanic consoles as shown in the Use Case Diagrams. Also, if the owner were to login to the Employee Console, he would see an expanded selection of Controllers because he has more access privileges.

Each Console is instantiated with an ArrayList of Controllers that represent valid operations on a particular console.

Controllers are invoked with their start() method, and subcontrollers are invoked from their controllers with their execute() method, in a utilization of the **Command** pattern.

Finally, each Subcontroller's job is to accept as parameters a RentalLog and a Console and return an updated RentalLog to the controller that invoked it.

Using the Template Pattern for RentBike



The reuse of Subcontrollers lends itself very well to the **Template** pattern. An abstract Controller RentBike was created. As the behavior of selecting a customer and confirming a reservation is the same regardless of whether the rental was employee- or customer-initiated, those subcontrollers are properties of RentBike.

RentBike's start() method is the template method. It would have code such as:

```
public void start() {
      RentalLog log = makeBikeSelection(employeeConsole, log);
      log = selectCustomer.execute(employeeConsole, log);
      log = confirmReservation.execute(employeeConsole, log);
      employeeConsole.reset();
}
```

The two concrete subclasses would define "makeBikeSelection(employeeConsole, log)" differently.

```
// Subclass RentBikeFromQueue
public void makeBikeSelection(employeeConsole, log) {
      return selectRentalLog.execute(employeeConsole, log);
}

// Subclass RentBikeEmployee
public void makeBikeSelection(employeeConsole, log) {
      return selectBikes.execute(employeeConsole, log);
}
```

Queues

Queue Instantiation

The three "To Be" buttons ("To Be Rented" and "To Be Settled" on the Employee Console, and "To Be Inspected" on the Mechanic Console) execute the RentBikeFromQueue, SettleBikeReturn and InspectBike controllers respectively.

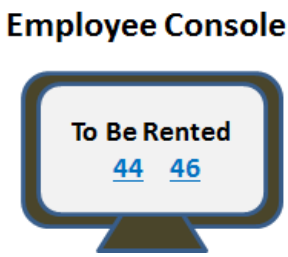
The SelectRentalLog subcontroller is coupled with the abstract Queue class. As Subcontrollers are instantiated by Controllers, each Controller that uses SelectRentalLog would create the appropriate type of Queue as seen below:

```
public RentBikeFromQueue() {
      SelectRentalLog selectRentalLog =
            newSelectRentalLog(RentalQueue.getQueueInstance());

public InspectBike() {
      SelectRentalLog selectRentalLog =
            newSelectRentalLog(InspectionQueue.getQueueInstance());

public SettleBikeReturn() {
      SelectRentalLog selectRentalLog =
            newSelectRentalLog(SettementQueue.getQueueInstance());
```

In this way, the SelectRentalLog subcontroller is decoupled from the specific type of queue. The Controller that creates it maintains the knowledge of which Queue to use. Since the Bike Shop has only one RentalQueue, InspectionQueue and SettlementQueue, static methods getQueueInstance() are used to ensure that the same queues are used throughout the application (**Singleton** pattern again). Now the same SelectRentalLog logic can be used for all three types of Queues.

The Controllers which need Queues (i.e., ReturnBike, SettleBikeReturn, SelectBikeCustomer and InspectBike – the latter of which needs two Queues), would be instantiated by the client (driver) class using the same technique.

Queues as Observables



The consoles which need to display items in a queue, i.e., the Employee Console and the Mechanic Console, are observers of particular queue objects. For example, the Employee Console observes the RentalQueue to see if new Rental logs have been added (thereby adding a selectable log to the console's display) or if rental logs have been processed and removed from the queue (thereby removing the selectable log from the console's display). This is an implementation of the **Observer** pattern.

In this example, the employee has selected the controller RentBikeFromQueue. That controller leverages subcontroller SelectRentalLog (see above) to display the contents of the rental queue (e.g.. rental log 44 and rental log 46) and receive the selected rental log the employee wishes to

process. As the Employee Console and the Mechanic Console are observers, their update() method will add or remove logs appropriately.

In a similar manner, the Employee Console observes the SettlementQueue to see which logs contain returned bikes that have completed their inspection, and the Mechanic Console observes the InspectionQueue to see which logs contain bikes that have just been returned.

Logging In and Employee Roles

Each console uses the static method authenticate() in the Authenticator class to validate login credentials. The abstract method getCredentials() in the abstract Console class needs to be implemented in all concrete subclasses. For now, a Customer Console will just return a special Credential object (userID = "public", password=null?) which guarantees automatic authentication. Customer consoles were set up this way, in case the Bike Shop wants to give certain customers database accounts at some later date and let them login to Customer consoles for future operations. This is also an implementation of the **Template** method, as the abstract Console knows how to login(), but delegates the behavior of getCredentials() to its subclasses.

Roles are maintained in a hashmap at the Person level, where roles correspond to Controllers. (The key of the roles hashmap is the class name of the Controller). After login, each Console knows the operator who has logged in based on the return value of the authenticate() method and can determine which operations (controllers) are available to that operator based on the roles hashmap.

Each console also was instantiated with an ArrayList of appropriate Controllers. Thus each console's display() method trivially loops through its allowable operations and its operator's privileges (roles). For example, the Mechanic Console will be instantiated with InspectBike, but if a Sales Rep logs into this console, that option will not be displayed. Similarly, if a Sales Rep logs into the Employee Console, the options only available to the owner will not be displayed.

Using States

For Rental Log Cancellation

In the analysis phase, we abandoned the idea of a Controller for CancelReservation. Rather, we determined that the cancel behavior varied depending on the state of the Rental log. Rental logs have various states as seen in the corresponding state diagram. At the moment, there are two different states in which a rental log may be cancelled.

When a customer uses the Customer console to start the process of renting bikes, a log is created and placed in the InRentalQueue state. If the customer decides to cancel the reservations at this point, two operations must occur:

1) the bikes on the log must be made available for other renters
2) the log must be removed from the rental queue

However, if the transaction was employee-mediated, then the cancel function only has to perform the first operation above. Using the **Strategy** pattern in conjunction with the **State** pattern, each State has its own CancelBehavior. At the moment, cancellation is not permitted once the rental has been finalized. Thus there are only three distinct types of cancel behaviors: one for the InRentalQueue state, one for the BeingRented state, and a third behavior which does not allow a

log to be cancelled. With the flexibility of the **State** pattern, we allow for future variants. For example, the Bike Shop may one day allow rental logs in the AwaitingReturns state to be cancelled before the designated return date for a partial rental fee.

The code below demonstrates how a rental log is cancelled

```
//  Cancelling a log from any subcontroller
private void cancelRentalLog(RentalLog log) {
      log.cancel();
}

//  The cancel method from a log
private void cancelRentalLog(RentalLog log) {
      state.cancel(this);
}

// The cancel() method from a State
public void cancel(RentalLog log) {
      cancelBehavior.cancel(log);
}

// The cancel() method in RemoveFromQueue (a CancelBehavior)
public void cancel(RentalLog log) {
      rentalQueue.remove(log);
      for (Detail d : log.getDetails()) {
            d.getRentable().setState(RentableState.AVAILABLE);
      }
}
```

It is important to note that the Subcontrollers are the agents which will set a rental log's state, rather than having each state set its subsequent state. This variation of the State pattern made sense to us, as it is the job of each Subcontroller to receive a log and return an updated log to its Controller. Setting the state in the Subcontroller felt cohesive.

Other behaviors might vary depending on the state. For example the printReceipt() method may print the log differently depending on its state.

For Bikes?

Bikes (as Rentables) also have a state, but it is currently datatyped as an enum rather than a State. Unlike rental logs, bikes did not have any methods whose implementation varied based on the bike state. However, were this to change in the future, the Rentable class could alter the datatype of its property *state* to be State leveraging the techniques used for rental logs.

Exercising the Model

To illustrate how all the components work together, the following steps would be taken for a customer-mediated rental.

1. The client instantiates all necessary classes including the Customer Consoles, the Employee Consoles, the Controllers, the Queues…

2. The client invokes the screen display method on each console to put them in login mode.
3. Template method customerConsoleA.login() is executed. Its implementation of getCredentials() allows a customer to use the console without a username and password.
4. The only Customer Console Controller (operations[0]) is displayed, using its label, as a GUI button.
5. The customer clicks the button which invokes the SelectBikeCustomer controller's start() method, and passes this method the designated console (customerConsoleA).
6. The SelectBikeCustomer controller's start() method fires, which:
    a) Creates a new rental log;
    b) Invokes the SelectBike subcontroller's execute() method, passing it customerConsoleA and the newly created rental log. The subcontroller, then:
        1) Interacts with customerConsoleA until it has enough information to add the rental log details (bikes to be rented, duration, etc.) to the log.
        2) Sets the state of each bike to Rentable.RESERVED
        3) Returns the amended log to the SelectBikeCustomer controller.
    c) Adds the log to the rental queue and sets the state of the log to InRentalQueue
    d) Resets customerConsoleA for the next customer
7. Meanwhile, the observable rental queue notified its console observers that a new rental log was added.
8. Each employee console was observing the rental queue, so they can display that a new customer-generated rental log is available for processing. (Each console's update() method fires, which refreshes the display.)
9. A Sales Rep at employeeConsoleC clicks the button which invokes the RentBikesFromQueue controller. (Assume the sales rep had previously logged in, and all Sales Rep controllers are displaying as buttons.)
10. The RentBikesFromQueue controller's start() method fires, which:
    a) Invokes the SelectRentalLog subcontroller's execute method, passing it employeeConsoleC and a new rental log. The subcontroller, then:
        1) Interacts with employeeConsoleC until a rental log is selected from the Rental Queue by the employee.
        2) Removes the selected rental log from the RentalQueue
        (This, in turn, causes the RentalQueue to notify its observers that a log has been removed, which results in the removal of this rental log from the displays of the other employee consoles.)
        3) Returns the selected rental log to the RentBikesFromQueue controller.
    b) Invokes the SelectCustomer subcontroller's execute method, passng it employeeConsoleC and the selected rental log. This subcontroller, then:
        1) Interacts with employeeConsoleC until a customer record is obtained
        2) Adds the customer to the log, applying any customer discounts to the log's details.
        3) Returns the amended log to the RentBikesFromQueue controller.
    c) Invokes the ConfirmReservations subcontroller's execute method, passing it employeeConsoleC and the amended rental log. This subcontroller, then:
        1) Interacts with employeeConsoleC until the employee finalizes the reservation after receiving payment.
        2) Amends the rental log with the Employee that waited on the customer
        3) Sets the state of each bike to Rentable.RENTED
        4) Sets the state of the rental log to AwaitingReturns

5) Returns the amended log to the RentBikesFromQueue controller

d) Resets employeeConsoleC, so the employee can process the next request.

Other Built-in Flexibilities

- The LogDetail class has a reference to the abstract class Rentable, so that it will not have to change if the Bike Shop suddenly decides to rent Jet Skis.
- Both the EmployeeConsole and the MechanicConsole were subclasses of abstract super class ObserverConsole. ObserverConsole is the class that forces the implementation of the update() method.
- Rental logs have a reference to abstract Employee to allow for either the Owner or the Sales Rep to process rentals (and maybe someday the Mechanic).

CLASS DIAGRAM

Diagram 1: People, Consoles and Queues

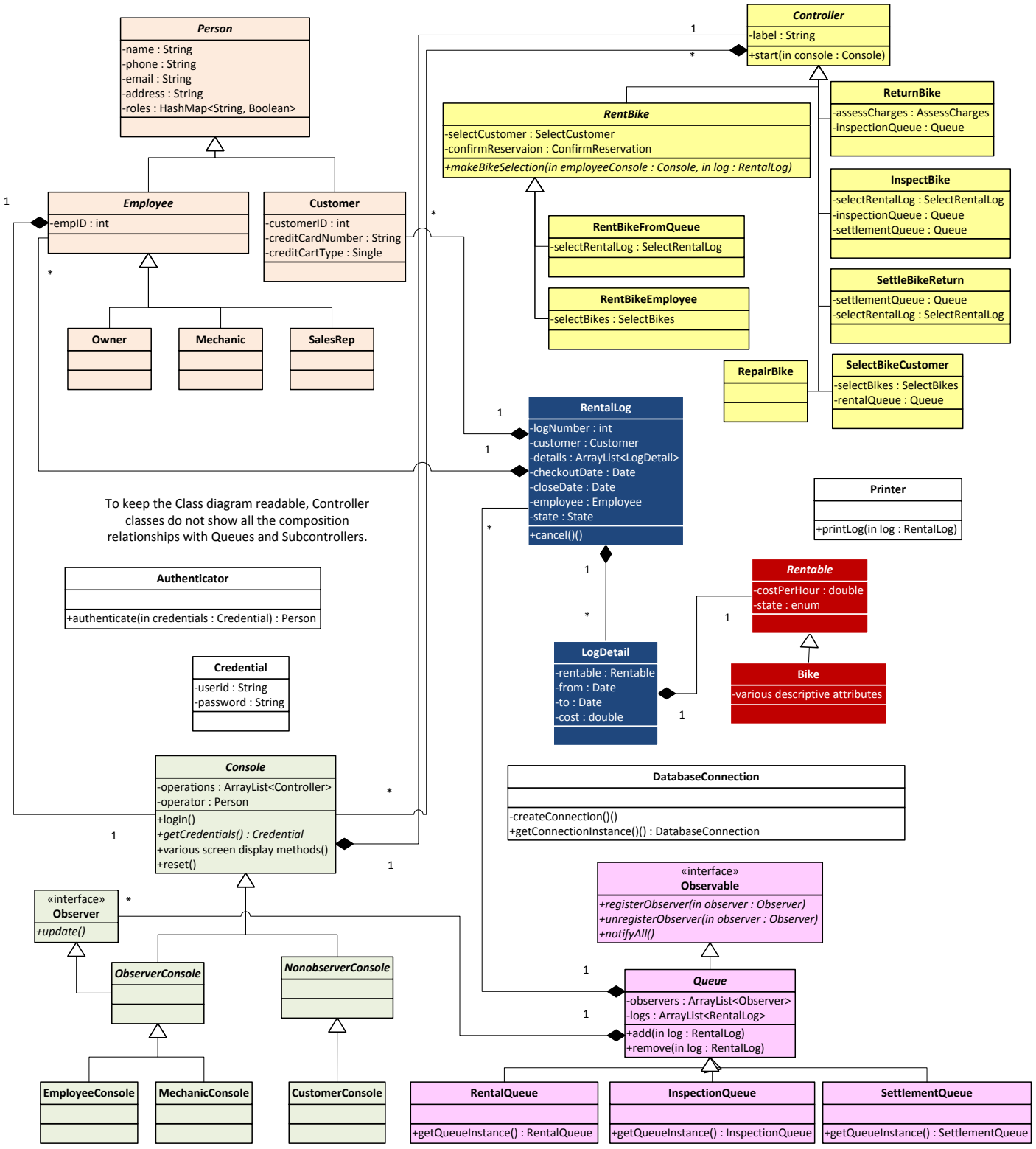
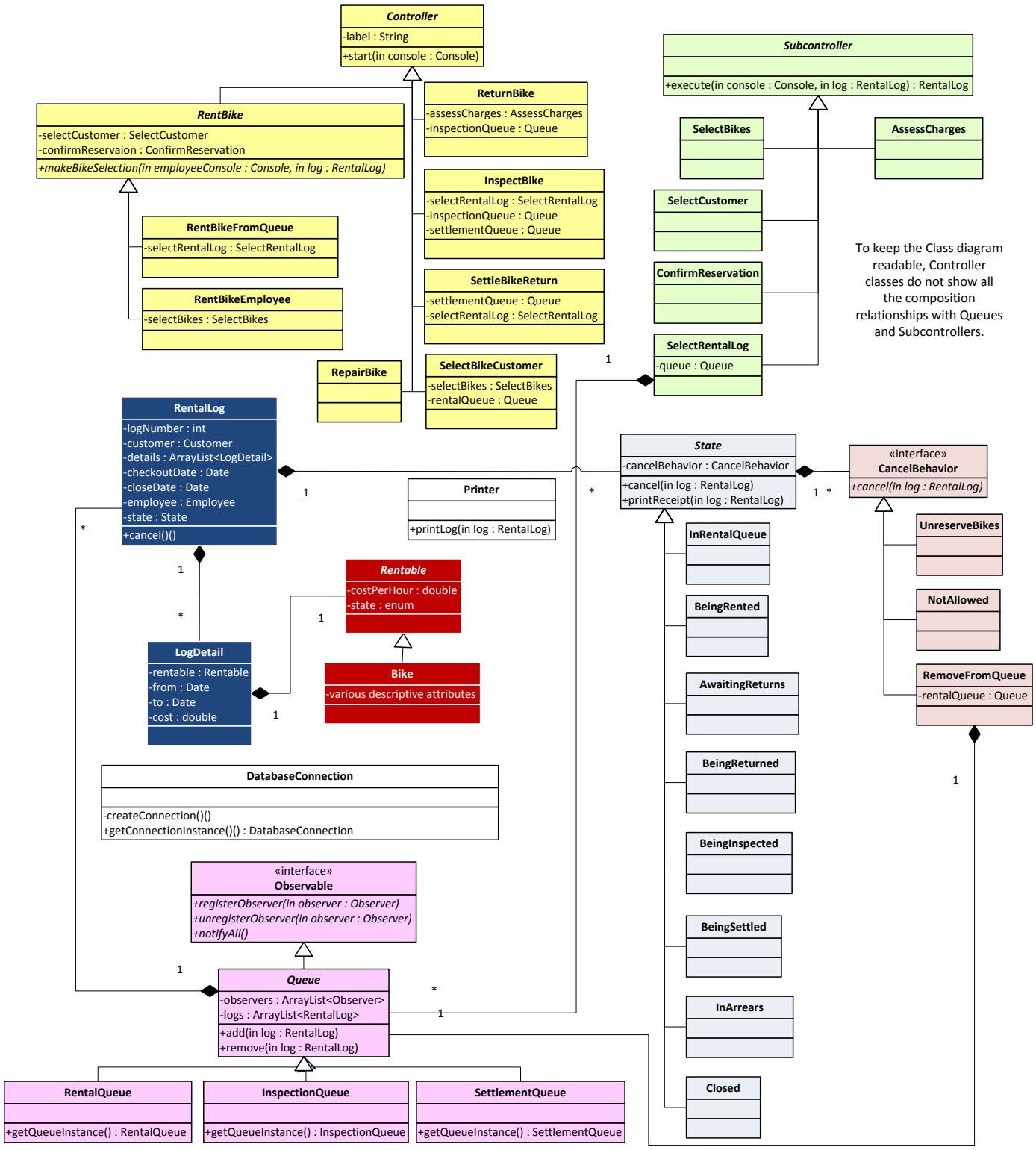


Diagram 2: Controllers, Subcontrollers and States



CRC CARDS

Notes:
- CRC Cards will not include DatabaseConnection as a Collaborator since they are all likely to be using it.
- CRC Cards will not show collaborators inherited from superclasses

Controllers and Subcontrollers

Abstract Class Controller

Responsibilities	**Collaborators**
Require all subclasses to have a method to start the controller action Provide a way for Consoles to provide a representation to users	Console

Abstract Class RentBike

Responsibilities	**Collaborators**
Provide common properties between operations required to rent bikes from the employee console only	SelectCustomer ConfirmReservation

Class SelectBikeCustomer

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to select bikes at the customer console Add customer's RentalLog to RentalQueue	SelectBikes RentalQueue RentalLog

Class RentBikeFromQueue

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to rent bikes that have been placed into the queue by customers	SelectRentalLog SelectCustomer ConfirmReservation RentalLog RentalQueue

Class RentBikeEmployee

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to rent bikes from the employee console	SelectBikes SelectCustomer ConfirmReservation RentalLog

Class ReturnBike

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to return bikes from customers at the employee console	AssessCharges RentalLog InspectQueue

Class InspectBike

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to inspect bikes being returned at the mechanic console	RentalLog InspectQueue SettlementQueue

Class SettleBikeReturn

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to settle payment after bikes have been inspected	RentalLog SettlementQueue

Class RepairBike

Responsibilities	**Collaborators**
Coordinate the console and subcontrollers that are needed to start the repair bike process	Bike

Abstract Class Subcontroller

Responsibilities	**Collaborators**
Updates a RentalLog as more information is obtained Use a console to get information from the user	RentalLog Console

Class SelectBikes

Responsibilities	**Collaborators**
Find all bikes that match user description Record proposed rental dates for those bikes Create a new rental log with all of the information the user has submitted	Bike

Class SelectCustomer

Responsibilities	**Collaborators**
Select a customer to be associated with the rental log	Customer

Class ConfirmReservation

Responsibilities	Collaborators
Update rental log state after employee has collected money	Employee
Print receipt	Printer
Update the state of the RentalLog	State

Class SelectRentalLog

Responsibilities	Collaborators
Select a RentalLog from the queue	RentalQueue
Update the state of the selected log	State

Class AssessCharges

Responsibilities	Collaborators
Determine which bike(s) are being returned	Bike
Determine the late fees for each bike	State
Update the state of the RentalLog	InspectionQueue

Queues

Interface Observable

Responsibilities	Collaborators
Provide a common set of methods that need to be implemented to make some object have subscribers to when it changes	Observer

Abstract Class Queue

Responsibilities	Collaborators
Notify Observers when elements have been added to this queue	RentalLog
Typical queue functions	Observer

Class RentalQueue

Responsibilities	Collaborators
Allow adding RentalLog objects to this queue	None
Notify Employee Consoles (observer) when a new object is added	

Class InspectionQueue

Responsibilities	Collaborators
Allow adding RentalLog objects to this queue Notify Mechanic Consoles (observer) when a new object is added	None

Class SettlementQueue

Responsibilities	Collaborators
Allow adding RentalLog objects to this queue Notify Employee Consoles (observer) when a new object is added	None

Rental Log and Details

Class RentalLog

Responsibilities	Collaborators
Encapsulate information about each rental	Customer LogDetail Employee State

Class LogDetail

Responsibilities	Collaborators
Encapsulate information about a particular bike rental	Bike

States

Abstract Class State

Responsibilities	Collaborators
Provide a common field to determine cancel behavior Provide statefulness for a RentalLog as it goes through rentals/returns	RentalLog CancelBehavior

Class InRentalQueue

Responsibilities	Collaborators
Encapsulate the actions that can occur when a RentalLog has been added to the RentalQueue	None

Class BeingRented

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog is in the process of being rented	None

Class AwaitingReturns

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog has been rented and is now awaiting returns	None

Class BeingReturned

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog is in the process of returning one or more bikes	None

Class BeingInspected

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog is in the process of being inspected	None

Class BeingSettled

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog is in the process of finishing a return	None

Class InArrears

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog has had all bikes returned, but the customer was unable to pay all fees at that time	None

Class Closed

Responsibilities	**Collaborators**
Encapsulate the actions that can occur when a RentalLog is no longer open for modification and no outstanding payments exist	None

People

Class Person

Responsibilities	**Collaborators**
Defines properties common to all people Provides common methods to all Person objects	None

Class Employee

Responsibilities	**Collaborators**
Define properties specific to Employees only Provide common methods to all Employee objects	None

Class Customer

Responsibilities	**Collaborators**
Define properties specific to Customers Provide common methods to Customer objects	None

Class SalesRep

Responsibilities	**Collaborators**
Define properties specific to SalesRep only Provide methods for SalesRep only	None

Class Owner

Responsibilities	**Collaborators**
Define properties specific to Owner only Provide methods for Owner only	None

Class Mechanic

Responsibilities	**Collaborators**
Define properties specific to Mechanic only Provide methods for Mechanic only	None

Consoles

Abstract Class Console

Responsibilities	**Collaborators**
Provide a common interface for all consoles to adhere to Provide common fields that all consoles need Require all subclasses to provide a way to login to the console	Authenticator Person Controller Credential

Interface Observer

Responsibilities	**Collaborators**
Provide a common framework for classes that must register with the Observables	None

Abstract Class ObserverConsole

Responsibilities	**Collaborators**
Provide an abstraction for any console that should be allowed to observe a queue to use	None

Class EmployeeConsole

Responsibilities	**Collaborators**
Provide an interface between the software system and the employees operating at an employee console Restrict access to different controllers (functions/flows) based on the operator type	None

Class MechanicConsole

Responsibilities	**Collaborators**
Provide an interface between the system and the mechanic Restrict access to non-mechanic employees	None

Interface NonObserverConsole

Responsibilities	**Collaborators**
Provides a common framework for classes that do not have to register with the Observables	None

Class CustomerConsole

Responsibilities	**Collaborators**
Provide an interface between the system and the customer Public access	None

Rentable Items

Class Rentable

Responsibilities	**Collaborators**
Encapsulates the similarities between any item that can be rented from the bike shop	None

Class Bike

Responsibilities	**Collaborators**
Represents a bike in the bike shop	None

Authentication Classes

Class Credential

Responsibilities	**Collaborators**
Encapsulates the users attempt (user input) to log into a console	None

Class Authenticator

Responsibilities	**Collaborators**
Determines whether or not an attempted login at any console is valid or not	Credential

Behaviors

Interface CancelBehavior

Responsibilities	**Collaborators**
Provides an interface for each state to implement its own cancel behavior when a cancel action needs to happen	RentalLog

Class NotAllowedCancelBehavior

Responsibilities	**Collaborators**
This behavior is used by any state where a cancellation is not valid.	None

Class UnreserveBikesCancelBehavior

Responsibilities	**Collaborators**
This behavior is used by any state where a cancellation results in unreserving the bikes in the rental log	Bike

Class RemoveFromQueueCancelBehavior

Responsibilities	**Collaborators**
This behavior is used by any state where a cancellation results in both removing this rental log from the RentalQueue and unreserving all bikes in the Rental log	Bike RentalQueue

Miscellaneous

Class DatabaseConnection

Responsibilities	**Collaborators**
This class is used when any controller needs access to the Database	None

Class Printer

Responsibilities	**Collaborators**
Takes rental log data from the system and sends it to a physical printer	RentalLog