# Appendix B

# iCoot Case Study

## B.1 BUSINESS REQUIREMENTS

This section documents the business requirements modeling carried out during the requirements phase of the iCoot development, in terms of project mission statement and business use case model. The business use case model also applies to the full Coot system.

### B.1.1 Customer's Mission Statement

Below is the mission statement delivered by Nowhere Cars at the start of the Coot project:

> Since we automated the tracking of cars at our stores – using bar codes, counter-top terminals and laser readers – we have seen many benefits: the productivity of our rental assistants has increased 20%, cars rarely go missing and our customer base has grown strongly (according to our market research, this is at least partly due to the improved perception of professionalism and efficiency.
>
> The management feels that the Internet offers further exciting opportunities for increasing efficiency and reducing costs. For example, rather than printing catalogs of available cars, we could make the catalog available to every Internet surfer for browsing on-line. For privileged customers, we could provide extra services, such as reservations, at the click of a button. Our target saving in this area is a reduction of 15% in the cost of running each store.
>
> Within two years, using the full power of e-commerce, we aim to offer all of our services via a Web browser, with delivery and pick-up at the customer's home, thus achieving our ultimate goal of the virtual rental company, with minimal running costs relative to walk-in stores.

Working with the customer, this mission statement was expanded into business use cases.

### B.1.2 Actor List

- Assistant: An employee at one of our stores who helps a Customer to rent a Car and reserve a CarModel.
- Customer: A person who pays us money in return for one of our standard services.

- Member: A Customer whose identity and credit-worthiness have been validated and who, therefore, has access to special services (such as making a Reservation by phone or over the Internet).
- NonMember: A Customer whose identity and credit-worthiness have not been checked and who, therefore, must provide a deposit to make a Reservation and surrender a copy of their License to rent a Car.
- Auk: The existing system that handles Customer details, Reservations, Rentals and the Catalog of available CarModels.
- DebtDepartment: The department that deals with unpaid fees.
- LegalDepartment: The department that deals with accidents in which a rented Car has been involved.

## B.1.3 Use Case List

- B1:Customer Rents Car: Customer rents a Car that they have selected from those available.
- B2:Member Reserves CarModel: Member asks to be notified when a CarModel becomes available.
- B3:NonMember Reserves CarModel: NonMember pays a deposit to be notified when a CarModel becomes available
- B4:Customer Cancels Reservation: Customer cancels an unconcluded Reservation, by phone or in person.
- B5:Customer Returns Car: Customer returns a Car that they have rented.
- B6:Customer Told CarModel Is Available: Customer is contacted by an Assistant when a Car becomes available.
- B7:Car Reported Missing: Customer or Assistant discovers that a Car is missing.
- B8:Customer Renews Reservation: Customer renews a Reservation that has been outstanding for more than a week.
- B9:Customer Accesses Catalog: Customer browses the catalog, in-Store or at home.
- B10: Customer Fined for Uncollected Reservation: Customer fails to collect a Car that they have reserved.
- B11:Customer Collects Reserved Car: Customer collects a Car that they have reserved.
- B12:Customer Becomes a Member: Customer provides CreditCard details and proof of address to become a Member.
- B13:Customer Notified Car Is Overdue: Assistant contacts Customer to warn them that a Car they have rented is more than a week overdue.
- B14:Customer Loses Keys: Replacement keys are provided for a Customer who has lost them.
- B15:MembershipCard Is Renewed: Assistant contacts Member to renew membership when their CreditCard has expired.
- B16:Car Is Unreturnable: A Car is wrecked or breaks down.

## B.1.4 Use Case Communication Diagrams

Communication diagrams were not used widely during business requirements modeling (although they were used extensively during system requirements gathering). However, one diagram (see Figure B.1) was produced to illustrate the external and internal actors involved in B3:NonMember Reserves CarModel.

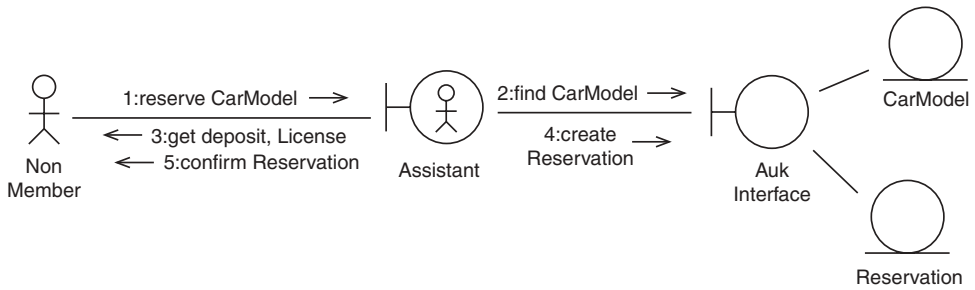


**Figure B.1: Communication diagram for B3:NonMember Reserves CarModel**

## B.1.5 Use Case Activity Diagrams

Activity diagrams were not used widely during the business requirements modeling. However, one diagram (see Figure B.2) was produced to illustrate the finer points of the B3:NonMember Reserves CarModel use case.

## B.1.6 Use Case Details

B1:Customer Rents Car.

1. Customer tells Assistant which CarModel they'd like to rent.
2. If Auk indicates no such Car is available, Customer is offered an alternative.
3. If there is a Car available, Assistant marks the Car as taken in Auk.
4. Assistant asks for Customer's License to confirm their identity.
5. For a Member, Assistant takes their number from their MembershipCard and checks that they have no outstanding fees and that they have not been barred.
6. For a NonMember, Assistant checks whether they're already in Auk; if they're not, Assistant scans a copy of their License into Auk, and records their name, phone number and license number.
7. If Customer's details are satisfactory and they have paid any outstanding fees, they're charged for the Rental.
8. If the payment fails, the Car is released in Auk.

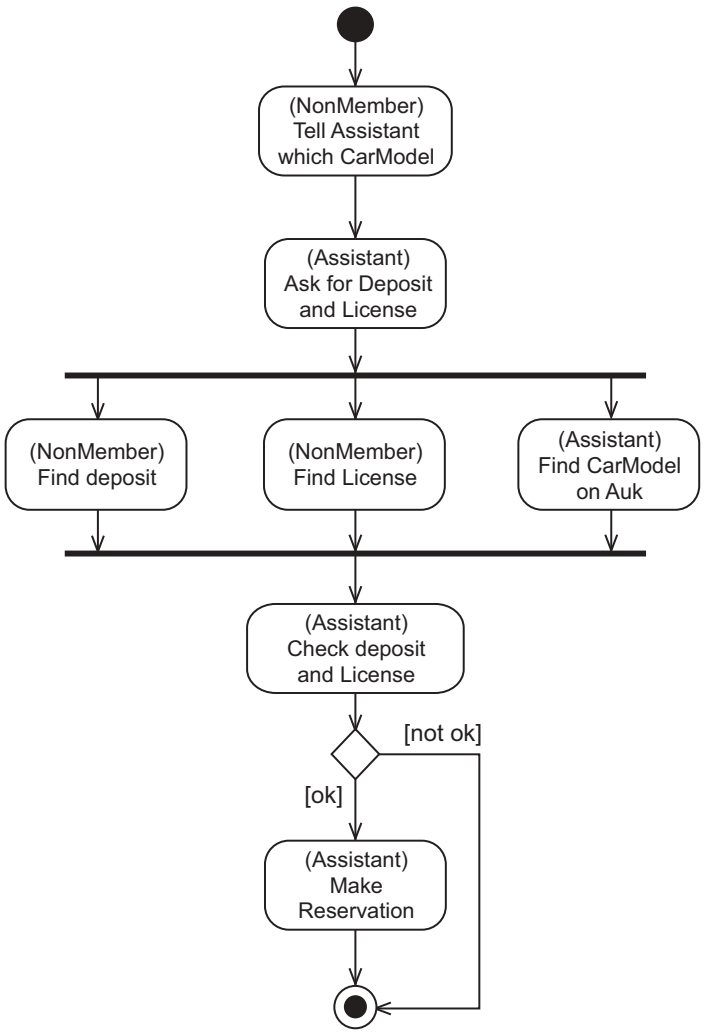


**Figure B.2: Activity diagram for B3:NonMember Reserves CarModel**

9. If the payment does not fail, the Customer is given the keys and directed to the display area.

B2:Member Reserves CarModel.

1. Member tells Assistant their membership number (over the phone or in person).
2. Member tells Assistant which CarModel to reserve.
3. If Member has not been barred, their CreditCard has not expired, and they have no outstanding fees, a Reservation is made on Auk.

4. If the Reservation is being made over the phone, Member can pay outstanding fees by confirming their CreditCard details, which must match those stored in Auk and must not have expired.
5. Member is told the reservation number.

B3:NonMember Reserves CarModel.

1. NonMember tells Assistant which CarModel to reserve.
2. Assistant finds CarModel on Auk.
3. Assistant asks for a deposit for the Reservation.
4. Assistant asks for NonMember's License and phone number.
5. Assistant checks License visually.
6. If License looks valid, Assistant creates a new Reservation, recording the License number, phone number and a scan of the License in Auk.
7. Assistant gives NonMember a Reservationslip containing the unique reservation number.

B4:Customer Cancels Reservation.

1. At any time, Customer can cancel a Reservation.
2. Member objects can do this over the phone or in person, by providing their membership number.
3. NonMembers must cancel in person: they present their License to an Assistant, who checks that it matches the scan in Auk, and refunds their deposit.
4. If a Car has already been moved to the reserved area, a matching Car is moved back to the display area.

B5:Customer Returns Car.

1. When a Car is returned to the check-in area, Assistant scans bar code to confirm the return and checks that the tank is full.
2. Car is returned to the display area by an Assistant.
3. If Customer returns an overdue Car or a Car with a tank that is not full, Customer must pay the appropriate amount – Members can do this using their existing credit card details, if they have not expired.
4. If the Customer refuses to pay, their details are passed to the DebtDepartment.

B6:Customer Told Car Model is Available.

1. When a Car is returned, Auk tells Assistant whether it matches any Reservation objects.
2. If so, Assistant moves Car to the reserved area.
3. On a first-come-first-served basis, Assistant will try to contact a matching Customer by phone.
4. If a Customer can't be reached within two days, their Reservation is canceled and the Car is moved out of the reserved area to the display area.

B7:Car Reported Missing.

1. If a Car that Auk indicates is in the display area can't be found when it is needed or during a stock check, Car is reported stolen to the police.
2. If a Car is reported missing by a Customer, it is reported stolen to the police, along with License details of the Customer (as the last known keeper of the vehicle).
3. In both cases, the date of loss is recorded on Auk.

B8:Customer Renews Reservation.

1. If a Reservation can't be satisfied within seven days, the Reservation must be renewed.
2. Assistant has two days to contact Customer by phone to see if they wish to renew the Reservation for a further seven days.
3. If the Customer doesn't wish to renew, the Reservation is canceled; Customer must return to the Store and present their License to retrieve their deposit.

B9:Customer Accesses Catalog.

1. Customers can come into the Store to browse a paper catalog.
2. For a fee, they can take a copy of the catalog home with them.
3. If they choose to join the mailing list, they will receive a free copy of the catalog by mail every six months.

B10: Customer Fined for Uncollected Reservation.

1. If a CarModel has become available for a particular Reservation and an Assistant told the Customer by phone that it's available, Customer has two days to collect.
2. If Customer fails to collect, the Reservation is concluded and an Assistant moves a matching Car from the reserved area back to the display area.
3. For NonMembers, their deposit is forfeited.
4. For Members, a fine is recorded on Auk and their details are passed to the DebtDepartment.

B11:Customer Collects Reserved Car.

1. Customer comes to the Store to collect a Car from the reserved area.
2. Customer presents License.
3. If the License matches the details on Auk, the Reservation is marked as concluded.
4. An Assistant gives the keys to the Customer and directs them to the reserved area.

B12:Customer Becomes Member.

1. In order to become a Member, Customer must offer their License, further proof of address, and a credit card.
2. Assistant checks License and proof of address.
3. Assistant checks CreditCard with CreditCardCompany.

4. If okay, Assistant records License number, address, phone number and CreditCard details in Auk.
5. Auk issues new MembershipCard with unique membership number.
6. If the CreditCard expires, no further member actions are permitted unless the member returns to the Store to show a new CreditCard.

B13:Customer Notified Car is Overdue.

1. Since a Rental is paid up-front, Customer is warned if they have forgotten to return a Car.
2. If Car is more than one week overdue, an Assistant will attempt to contact Customer by phone.
3. If Customer can't be contacted for two weeks, Car is reported missing (see B7).

B14:Customer Loses Keys.

1. If Customer notifies Assistant that they have lost keys, replacement keys are provided, by courier if necessary.
2. Cost of replacement is added to Customer's details in Auk.

B15:MembershipCard is Renewed.

1. Auk records that Member whose CreditCard has expired is not in good standing.
2. Auk informs Assistant that Member's Credit card has expired.
3. Assistant contacts Member by phone to tell them that they must renew their membership.
4. Member returns to Store with fresh CreditCard and details are entered into Auk.
5. Auk records that Member is in good standing.

B16:Car is Unreturnable.

1. If Customer tells Assistant that Car is wrecked or breaks down, Assistant arranges recovery.
2. If Car is wrecked, details are passed to LegalDepartment.

# B.2 SYSTEM REQUIREMENTS

This section documents the results of system modeling during the requirements phase of the iCoot development, in terms of user interface sketches and a system use case model.

## B.2.1 User Interface Sketches

The user interface sketches for iCoot, produced with the help of the customer, are shown in Figures B.3 through B.10.

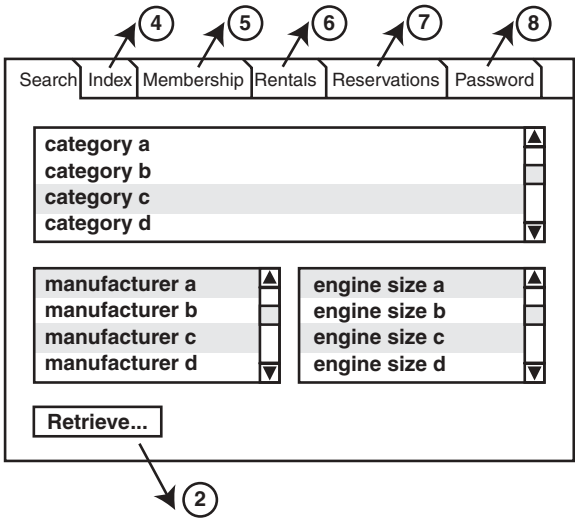


**Figure B.3: User interface sketch 1 (creating a query)**

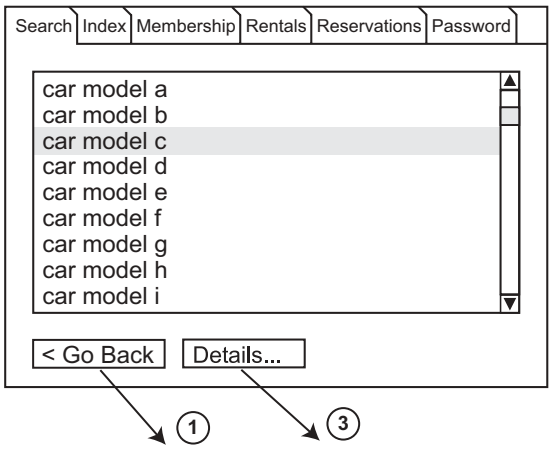


**Figure B.4: User interface sketch 2 (viewing results)**

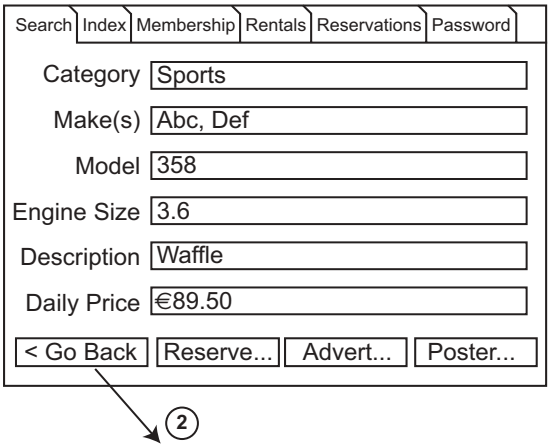


**Figure B.5: User interface sketch 3 (viewing car model details)**

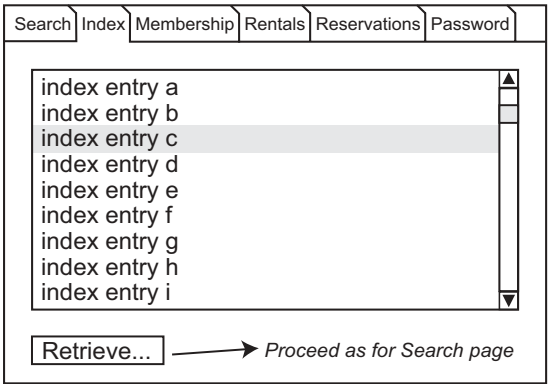


**Figure B.6: User interface sketch 4 (selecting an index heading)**

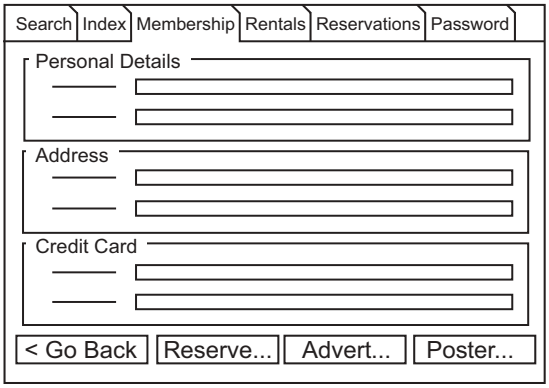


**Figure B.7: User interface sketch 5 (viewing membership details)**

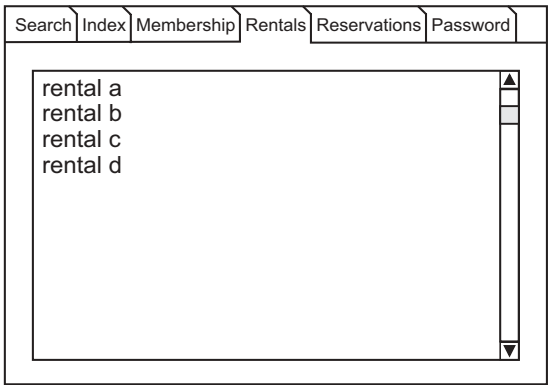


**Figure B.8: User interface sketch 6 (viewing rentals)**

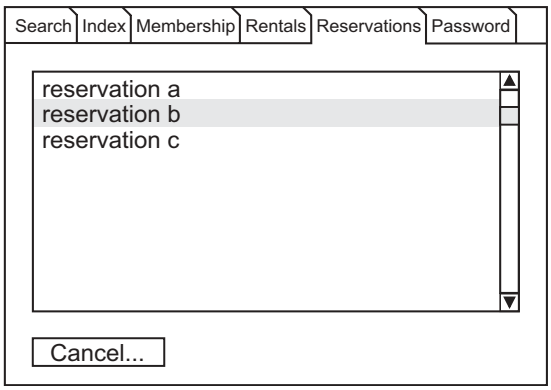


**Figure B.9: User interface sketch 7 (viewing reservations)**

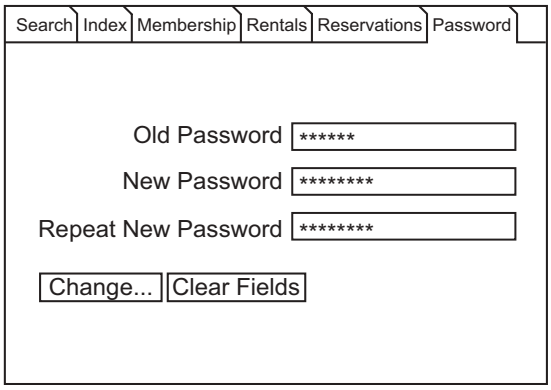


**Figure B.10: User interface sketch 8 (changing a password)**

## B.2.2 Actor List

- Customer: A person using a Web browser to access iCoot.
- Member: A Customer who has presented their name, address and CreditCard details at one of our Stores; each Member is given an Internet password to accompany their membership number. (Specializes Customer.)
- NonMember: A Customer who is not a Member. (Specializes Customer.)
- Assistant: An employee at a Store who contacts Members to tell them about the progress of their Reservations.

## B.2.3 Use Case List

- U1:Browse Index: A Customer browses the index of CarModels. (Specializes U13, includes U2.)
- U2:View Results: A Customer is shown the subset of CarModels that were retrieved. (Included by U1 and U4, extended by U3.)
- U3:View CarModel Details: A Customer is shown the details of a retrieved CarModel, such as description and advert. (Extends U2, extended by U7.)
- U4:Search: A Customer searches for CarModels by specifying Categories, Makes and engine sizes. (Specializes U13, includes U2.)
- U5:Log On: A Member logs on to iCoot using their membership number and current password. (Extended by U6, U8, U9, U10 and U12.)
- U6:View Member Details: A Member views some of the details stored by iCoot, such as name, address and CreditCard details. (Extends U5.)
- U7:Make Reservation: A Member reserves a CarModel when viewing its details. (Extends U3.)
- U8:View Rentals: A Member views a summary of the Cars they're currently renting. (Extends U5.)
- U9:Change Password: A Member changes the password that they use to log on. (Extends U5.)
- U10: View Reservations: A Member views summaries of their unconcluded Reservations, such as date, time and CarModel. (Extends U5, extended by U11.)
- U11:Cancel Reservation: A Member cancels an unconcluded Reservation. (Extends U10.)
- U12:Log Off: A Member logs off from iCoot. (Extends U5.)
- U13:Look for CarModels: A Customer retrieves a subset of CarModels from the catalog. (Abstract, generalized by U1 and U4.)

## B.2.4 Use Case Diagram

The use case diagram for iCoot is shown in Figure B.11.

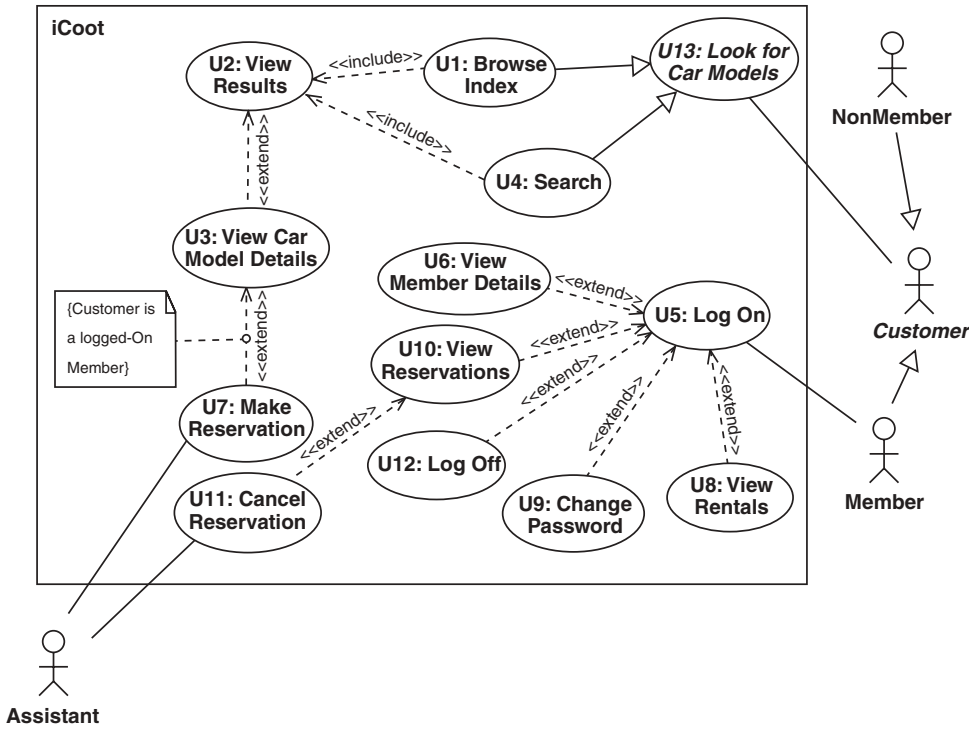


**Figure B.11: Use case diagram for iCoot**

## B.2.5 Use Case Survey

The use case survey for iCoot, describing how the use cases fit together, is:

Any Customer can look for CarModels in the catalog, by browsing the CarModel index (U1) or by searching (U4). In the latter case, the Customer specifies the Categories, Makes and engine sizes that they're interested in. Either way, after each retrieval, the Customer is shown the results as a collection of matching CarModels (U2), along with basic information such as CarModel name. The Customer can then choose to view extra information about particular CarModel objects such as a description and an advert (U3).

A Customer who has become a Member can log on (U5) and gain access to extra services. The extra services are: making a Reservation (U7), canceling a Reservation (U11), checking membership details (U6), viewing outstanding Reservations (U10), changing their log-on password (U9), viewing their outstanding Rentals (U8) and logging off (U12).

Assistants are involved in the life cycle of Reservations, moving Cars to and from the reserved area, for example.

Customers come in two varieties, Members and NonMembers.

Browsing the index and searching for CarModels are two different ways of looking for CarModels (U13). In order to view CarModel details, a Customer must be viewing the results of looking for models (via the browsing or searching route).

In order to reserve a CarModel, a Member must be viewing its details (NonMembers can't make reservations, even when they're viewing details).

In order to cancel a Reservation, a Member must be viewing their outstanding Reservations.

## B.2.6 Use Case Details

U1:Browse Index. (Specializes U13, includes U2.)
Preconditions: None.
1. Customer selects an index heading.
2. Customer elects to view CarModels for the selected index heading.
3. Include U2.

Postconditions: None.

U2:View Results. (Included by U1 and U4, extended by U3.)
Preconditions: None.
1. iCoot presents Customer with a summary of each retrieved CarModel, including model number and price.
2. Extend with U3.

Postconditions: None.

U3:View CarModel Details. (Extends U2, extended by U7.)
Preconditions: None.
1. Customer selects one of the matching CarModels.
2. Customer requests details of the selected CarModel.
3. iCoot displays details for the selected car model (makes, engine size, price, description, advert and poster).
4. If Customer is a logged-on Member, extend with U7.

Postconditions: iCoot has displayed details of selected CarModels.
Non-Functional Requirements: r1. Adverts should be displayed using a streaming protocol rather than requiring a download.

U4:Search. (Specializes U13, includes U2.)
Preconditions: None.
1. Customer selects required categories (if any).
2. Customer selects required Makes (if any).

3. Customer selects required engine sizes (if any).
4. Customer initiates the search.
5. Include U2.

Postconditions: None.

Abnormal paths: a1. If Customer specifies no categories, makes or engine sizes, rather than retrieving the entire catalog, iCoot should not allow the search to be initiated.

U5:Log On. (Extended by U6, U8, U9, U10 and U12.)

Preconditions: Member has obtained a password from their local Store.

1. Member enters the membership number.
2. Member enters the password.
3. Since iCoot must enforce one logon for a Member, Member can choose to steal (invalidate and thus take over from) an existing session.
4. Member elects to log on.
5. Extend with U6, U8, U9, U10, U12.

Postconditions: Member is logged on.

Abnormal Paths: a1. If the membership number/password combination is incorrect, iCoot informs Member that one of the two is incorrect (for security, they're not told which one).

a2. If the membership number/password combination is correct, but Member is already logged on and they have not elected to steal, iCoot informs Member.

U6:View Member Details. (Extends U5.)

1. Member elects to view membership details.
2. Member is presented with membership details (name, address, status, amount owing, CreditCard details).
3. For security reasons, iCoot must display only the last four digits of the Member's CreditCard number.
4. iCoot informs Member that to correct details, they must contact their local Store.

Postconditions: Member has been presented with membership details.

U7:Make Reservation. (Extends U3.)

Preconditions: Customer is a Member who has logged on.

1. Member elects to reserve CarModel for the details on display.
2. iCoot asks Member for confirmation, issuing a warning that failure to collect a reserved CarModel will result in a fine.
3. Member confirms Reservation.
4. iCoot shows Member the Reservation number and indicates that Assistant will be in touch when a Car is available.
5. When an Assistant logs on to Coot, Assistant is given a list of Reservations that require action.

6. Assistant takes necessary action to progress Reservations (e.g. promoting to Collectable if a Car is available and moving the Car to the reserved area).

Postconditions: Any requested Reservations have been made.

Abnormal Paths: a1. If Member declines Reservation conditions, no Reservation is made.

U8:View Rentals. (Extends U5.)

Preconditions: None. Relationships: U5.

1. Member elects to view their Rentals.
2. iCoot presents Member with summary of each Car they currently have out for rent (including number plate and due date).

Postconditions: iCoot has presented Member with summaries of Cars currently rented.

U9:Change Password. (Extends U5.)

Preconditions: None.

1. Member elects to change password.
2. Member enters old password (which is obscured on screen).
3. Member enters new password (obscured).
4. Member enters new password again (for confirmation, also obscured).
5. Member initiates the change.
6. iCoot asks for confirmation (warning that new password must be memorable).
7. If Member confirms, password is changed.

Postconditions: Password has been changed.

Abnormal Paths: a1. If old password is incorrect or new passwords don't match, Member is informed (but not given details of the error, for security) and password is unchanged.

a2. If old passwords match but new password doesn't follow password rules (a mix of at least six letters and digits), Member is informed and password is unchanged.

U10: View Reservation objects. (Extends U5, extended by U11.)

Preconditions: None.

1. Member elects to view Reservations.
2. iCoot displays summaries of the Member's outstanding (unconcluded) Reservations (including number, state, timestamp and CarModel number).
3. Extend with U11.

Postconditions: Member has been presented with summary of outstanding Reservations.

U11:Cancel Reservation. (Extends U10.)

Preconditions: None.

1. Member selects a Reservation.
2. Member elects to cancel the Reservation.
3. iCoot asks for confirmation.

4. Member confirms that they wish to cancel the Reservation.
5. iCoot marks the Reservation as Concluded and updates Assistants' terminals accordingly.

Postconditions: Any canceled Reservations that were confirmed have been marked as Concluded.
Abnormal Paths: a1. If Member doesn't confirm a cancellation, iCoot takes no action.

U12:Log Off.
Preconditions: None.
1. Member elects to log off.
2. iCoot ends current session.
3. iCoot makes Member-only functions unavailable to Member.

Postconditions: Member is logged off.
Abnormal Paths: a1. For security reasons, a logged-on Member is logged off automatically if they do not interact with iCoot for ten minutes.

U13:Look for CarModels (Abstract, specialized by U1 and U4.)
Preconditions: None.
Postconditions: Customer has been presented with summaries of retrieved CarModels.

## B.2.7 Supplementary Requirements

s1. The client applet must run in Java PlugIn 1.2 (and later versions).
s2. iCoot must be able to cope with a catalog of 100,000 CarModels.
s3. iCoot must be able to serve 1,000,000 Customers simultaneously with no significant degradation in performance.

## B.2.8 Use Case Priorities

Below is the list of use case priorities for iCoot, with the scores that were used for the first increment.

- **Green**:
  - U1: Browse Index
  - U4: Search
  - U2: View Results
  - U3: View CarModel Details
  - U5: Log On
- **Amber**:
  - U12: Logoff.
  - U6: View Member Details
  - U7: Make a Reservation
  - U10: View Reservations

- **Red**:
  - U11: Cancel Reservation
  - U8: View Rentals
  - U9: Change Password

During the first increment, U6 was also completed. The other use cases were completed during the second increment.

# B.3 ANALYSIS

This section documents the results of the analysis phase of the iCoot development, in terms of analysis class model, a state machine for a Reservation and use case realization (communication diagrams). The Reservation state machine also applies to the full Coot system. The class model includes a few pieces from the full Coot schema, such as NonMember and dateLost.

## B.3.1 Class Diagram

The analysis class diagram for iCoot is shown in Figure B.12. Most of these classes also appear in the design class model (Section B.5), so their descriptions have been placed in the Glossary (Section B.8), to avoid repetition.

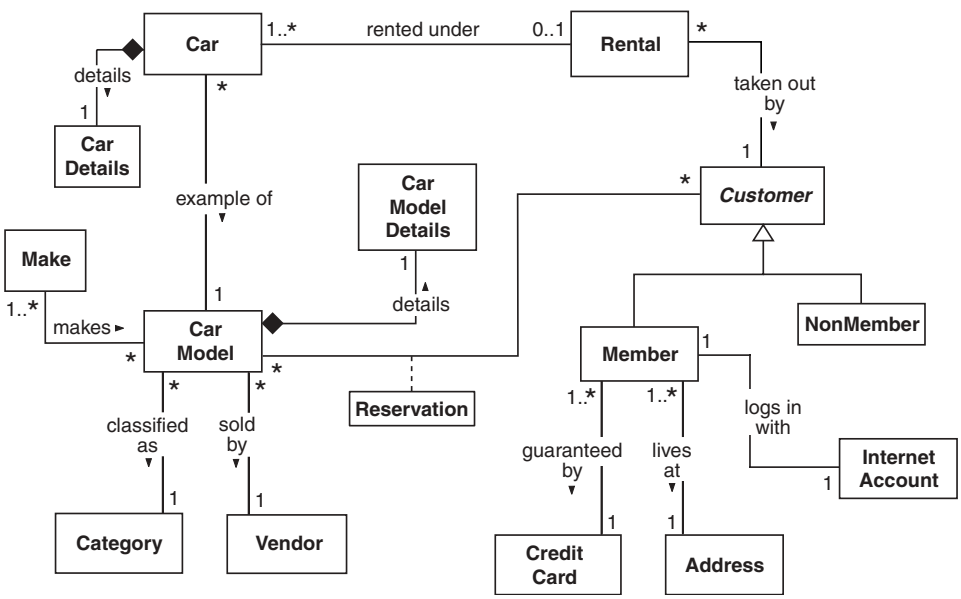


**Figure B.12: Analysis class diagram**

## B.3.2 Attributes

The class attributes for iCoot are shown in Figure B.13. These attributes also appear in the design class model as fields, where they're given types and descriptions – refer to the design documentation (Section B.5) for details.

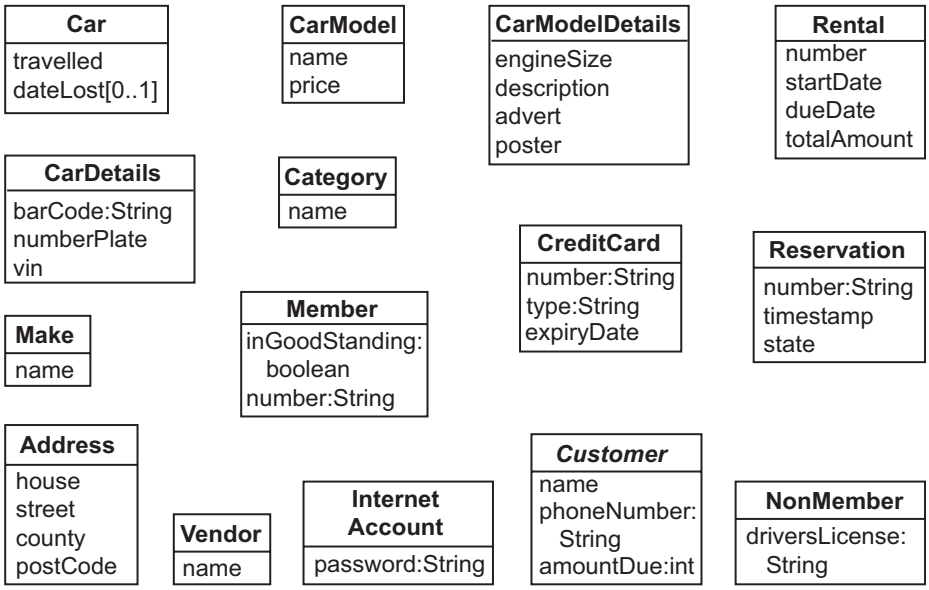


**Figure B.13: Analysis attributes**

## B.3.3 Operation List

- CarModel:
  - getSummary() – Fetch a summary of the receiver, including model number and price.
  - getDetails() – Fetch the receiver's details, including makes, engine size, price, description, advert and poster.
- CarModelHome:
  - findByIndexHeading(h:String) – Search for CarModel objects under index heading h.
  - findByQuery(categories,makes,sizes) – Search for CarModel objects with Category from categories, a Make in makes and engine size in sizes.
- LogonController:
  - logon(n:String,p:String,s:boolean) – Log on the Member with membership number n and password p, specifying whether or not to steal any existing session with s.

  - changePassword(m:Member,o:String,n1:String,n2:String) – Change the password for m to n1, as long as n2 matches and the current password is o.
  - logoff() – Log off the logged-on Member.
- Member:
  - getPassword():String – Fetch the receiver's password.
  - isLoggedOn():boolean – True if the receiver is logged on.
  - logon() – Log the receiver on.
  - logoff() – Log the receiver off.
  - getDetails() – Fetch the receiver's details, including name, address, status, amount owing and (concealed) credit card details
  - setPassword(p:String) – Set the receiver's password to p.
- MemberHome: findByMembershipNumber(n:String):Member – Find the Member with membership number m.
- MemberUI:
  - search(categories,makes,sizes) – Search for CarModel objects with a Category from categories, a Make in makes and engine size in sizes.
  - index(h:String) – Search for CarModel objects under index heading h.
  - logon(n:String,p:String,s:boolean) – Log on the Member with membership number n, password p, specifying whether or not to steal any existing session with s.
  - setMember(m:Member) – Set the logged-on Member to m.
  - showMemberDetails() – Show details for the logged-on Member.
  - showRentals() – Show Rental objects for the logged-on Member.
  - showReservations() – Show unconcluded Reservation objects for the logged-on Member.
  - changePassword(o:String,n1:String,n2:String) – Change the password for the logged on Member to n1, as long as n2 matches and the current password is o.
  - confirmChange() – Confirm that the password really should be changed.
  - reserve(c:CarModel) – Reserve c for the logged-on Member.
  - confirmReserve() – Confirm that the Reservation really should be made.
  - cancel(r:Reservation) – Cancel r.
  - confirmCancel() – Confirm that the Reservation really should be canceled.
  - showDetails(c:CarModel) – Show details for c.
  - logoff() – Log off the logged-on Member.
- NonMemberUI:
  - search(categories,makes,sizes) – Search for CarModel objects with a Category from categories with a Make in makes and engine size in sizes.
  - index(h:String) – Search for CarModel objects under index heading h.
- Rental: getSummary() – Fetch a summary of the receiver, including number plate and due date.
- RentalHome: findByMember(m:Member – Fetch the Rental objects for member m.

- Reservation:
  - getSummary() – Fetch a summary of the receiver, including number, timestamp, state and CarModel.
  - getNumber() – Fetch the receiver's number.
  - setState(s) – Set the receiver's state to s.
- ReservationHome:
  - findByMember(m:Member) – Fetch the reservations for m.
  - create(c:CarModel,m:Member) – Reserve c for m, with the current date and time.

## B.3.4 State Machine for a Reservation

Figure B.14 shows the state machine diagram for a Reservation, produced to model its complex life cycle. The accompanying state machine survey is:

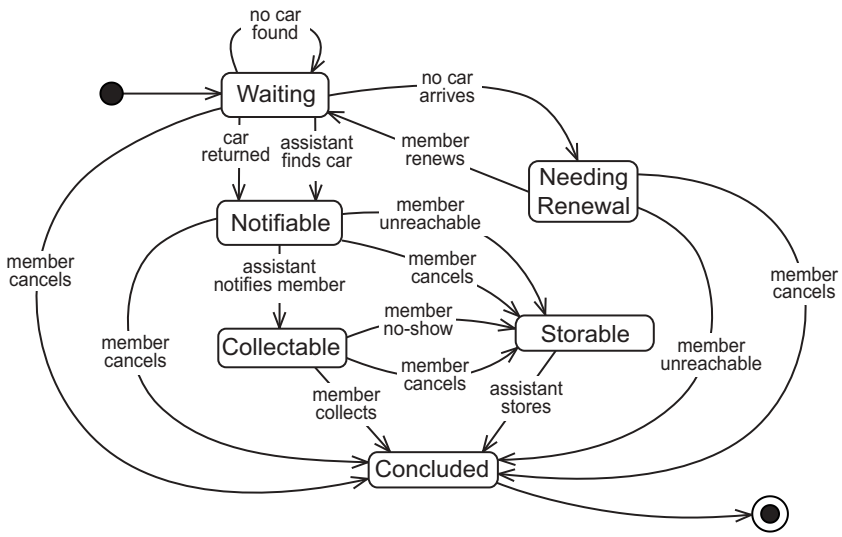


**Figure B.14: State machine diagram for a Reservation**

When a Member reserves a CarModel over the Internet, the Reservation is initially Waiting to be processed by an Assistant (this is so the Customer can make a Reservation without the intervention of an Assistant). The Reservation becomes Notifiable if, some time later, an Assistant finds a suitable unreserved Car in the display area of the car park, or if one is returned by a Customer. In this case, the Car is moved to the reserved area.

If no Car becomes available for a particular Reservation within a week, the Reservation becomes NeedingRenewal: the Member must be contacted, by phone or in person, so

that they can cancel the Reservation, or ask for it to be renewed for another week. If the Member cancels or can't be contacted within five days, the Reservation is Concluded.

Once a Reservation is Notifiable, the Member must be notified by an Assistant, in person or by phone, within three days; if the Customer can be reached, the Reservation is Collectable otherwise it becomes Displayable (a Car that was moved to the reserved area must be returned to the display area).

Once a Reservation is Collectable, the Member must collect the Car within three days: if they do collect, the Reservation is Concluded; otherwise, the Reservation becomes Displayable.

Once a Displayable Reservation's Car has been put back in the display area, the Reservation is Concluded.

At any time, the Member may cancel the Reservation over the Internet, by phone or in person.

The system will keep Assistants informed as to the state of current (not yet concluded) reservations, so that they can take appropriate action.

## B.3.5 Use Case Realization

The communication diagrams for iCoot, verifying the analysis class model, are shown in Figures B.15 through B.26, one per system use case. Note the use of **guards** (arbitrary conditions in brackets), to specify conditional messages and * to specify iteration (iteration guards can be used to control iteration, but these would have made the diagrams more complex).

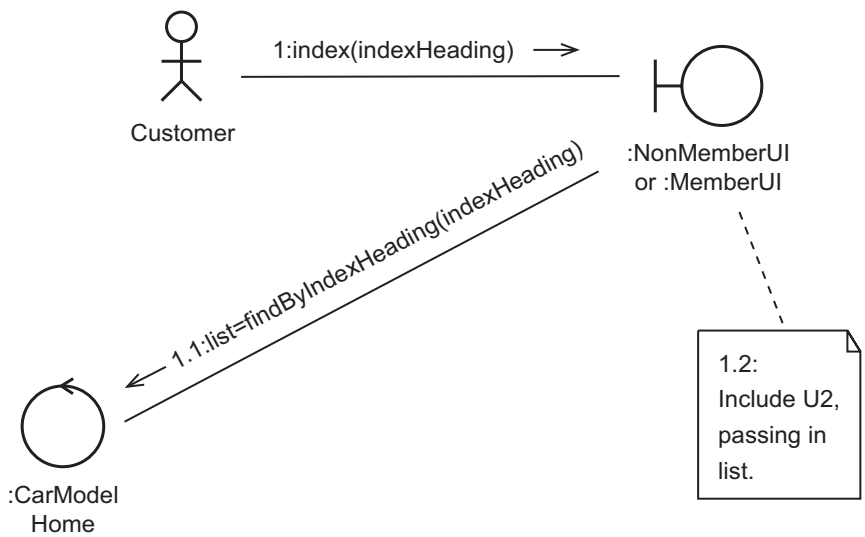


**Figure B.15: Communication diagram for U1:Browse Index**

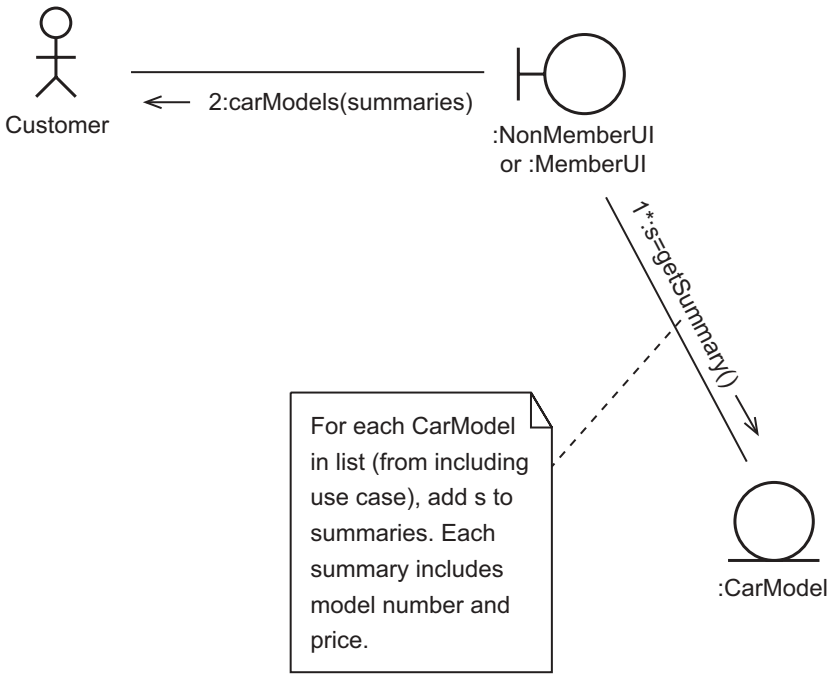


**Figure B.16: Communication diagram for U2:View Results**

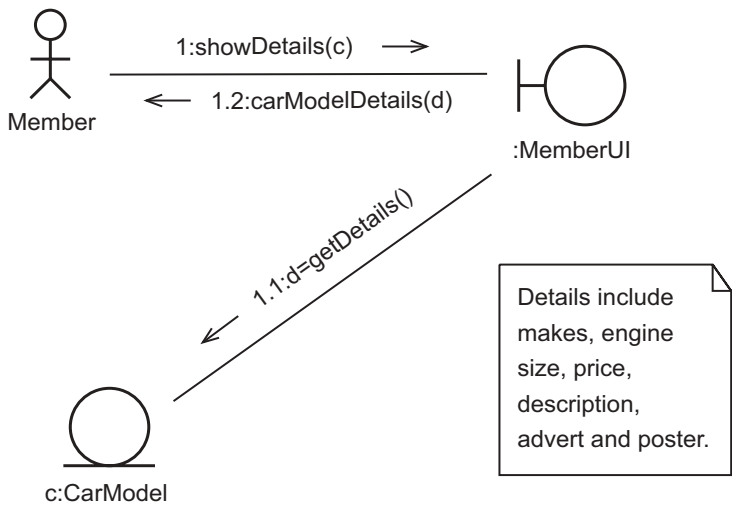


**Figure B.17: Communication diagram for U3:View CarModel Details**

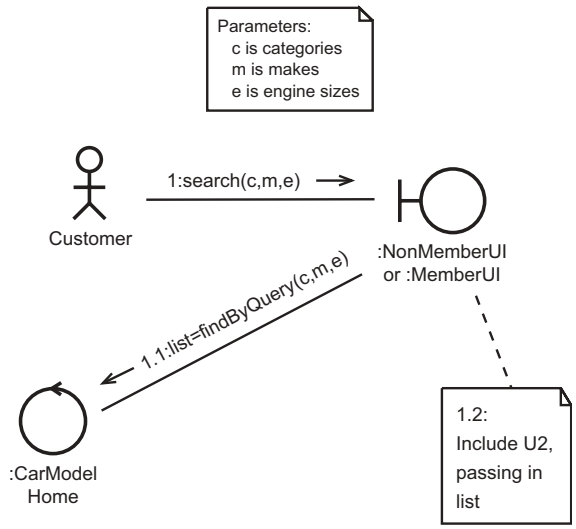


**Figure B.18: Communication diagram for U4:Search**

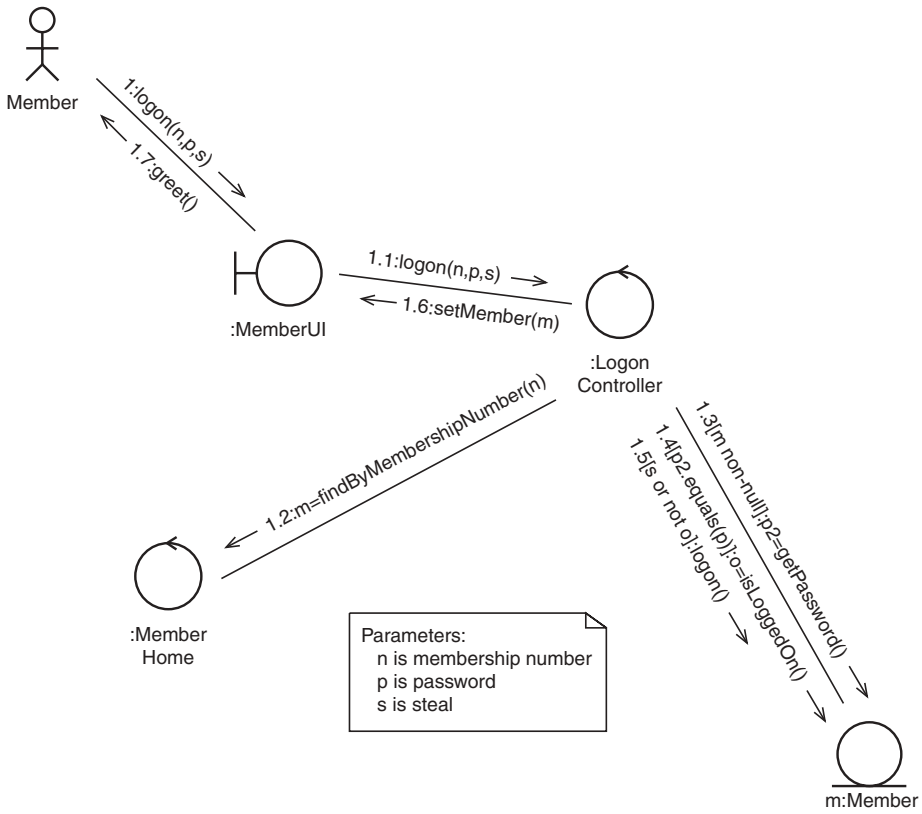


**Figure B.19: Communication diagram for U5:Log On**

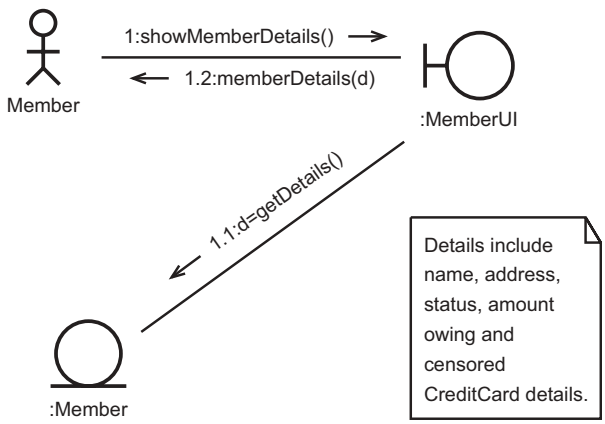


**Figure B.20: Communication diagram for U6:View Member Details**

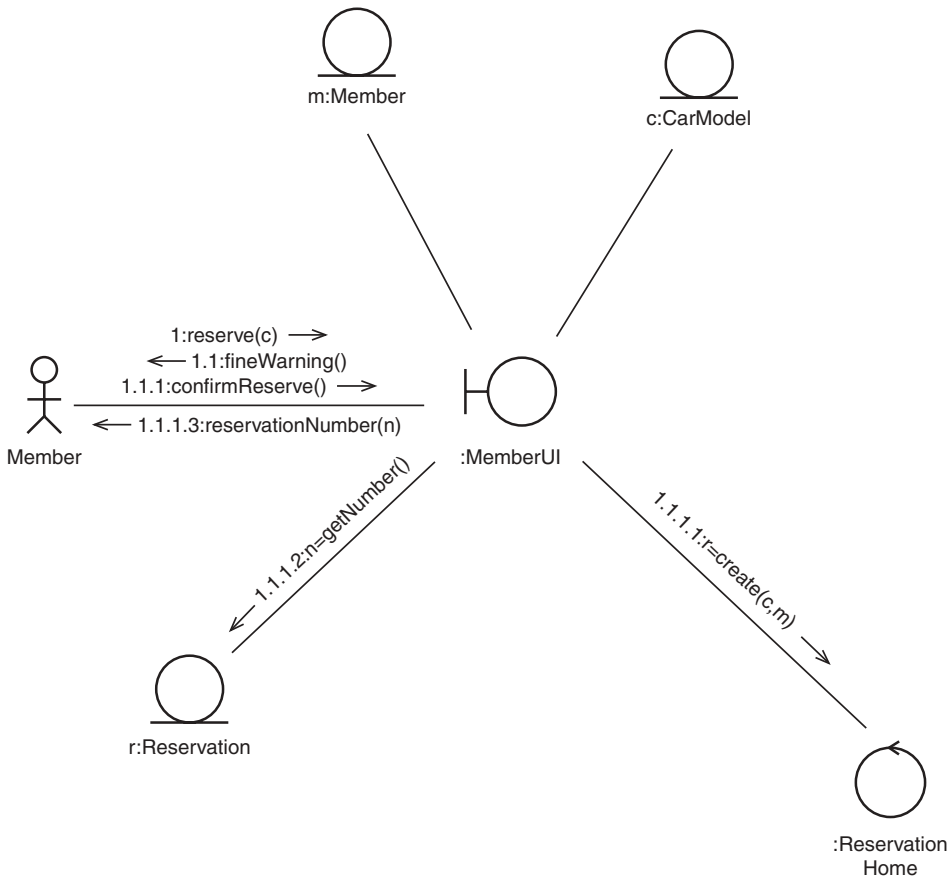


**Figure B.21: Communication diagram for U7:Make Reservation**

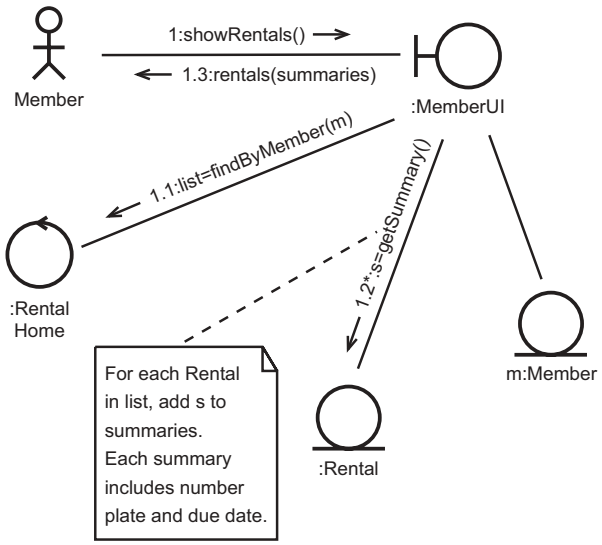


**Figure B.22: Communication diagram for U8:View Rentals**

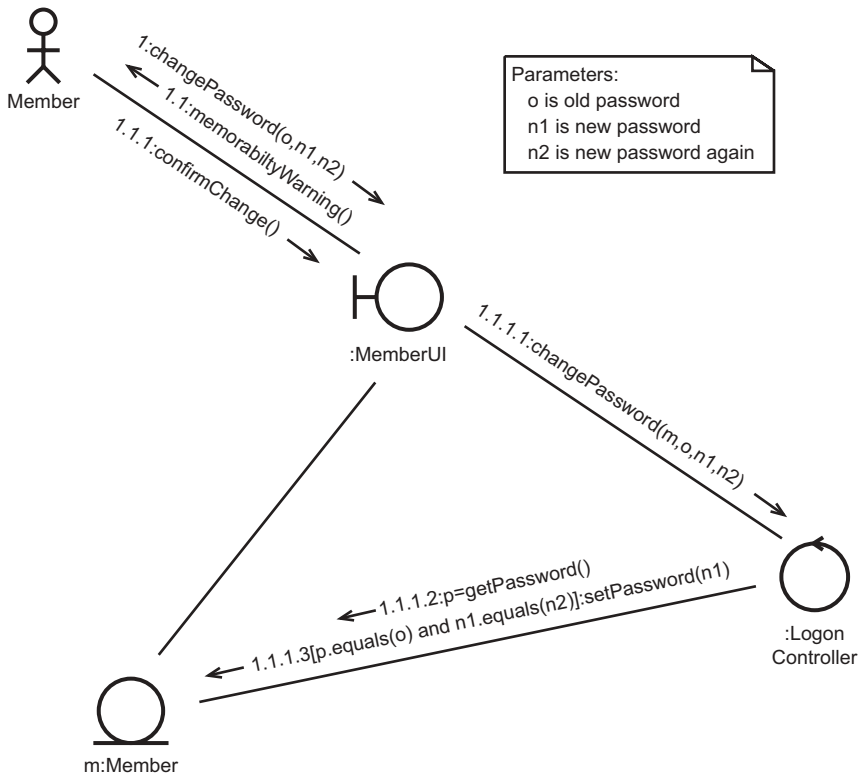


**Figure B.23: Communication diagram for U9:Change Password**

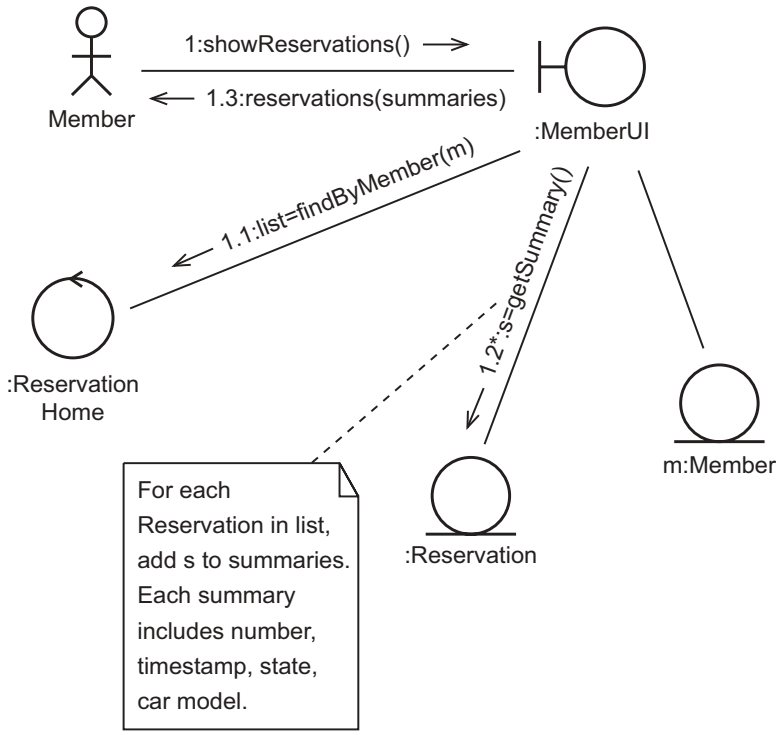


**Figure B.24: Communication diagram for U10: View Reservations**

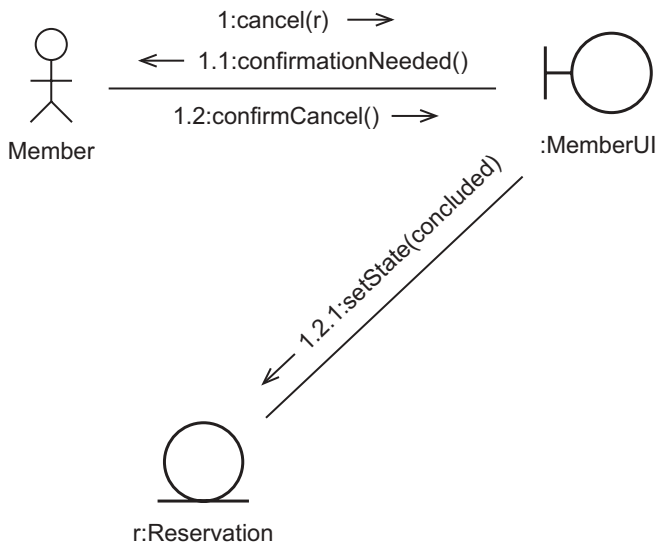


**Figure B.25: Communication diagram for U11:Cancel Reservation**

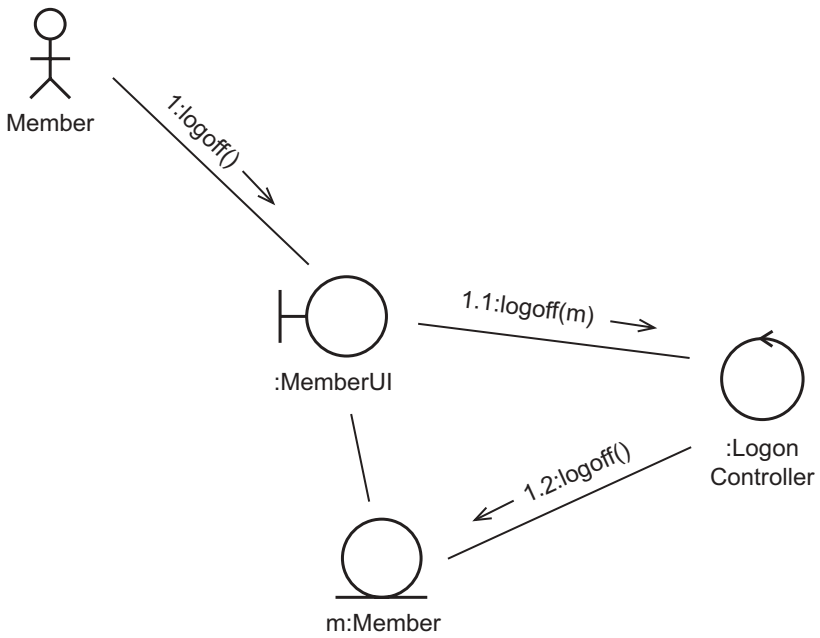


**Figure B.26: Communication diagram for U12:Log Off**

# B.4 SYSTEM DESIGN

This section documents the results of system design carried out during the design phase of the iCoot development, in terms of technology choices, layers, packages, deployment diagram, security policy and concurrency policy.

## B.4.1 Technology Choices

On the client side, the choice of technology is driven by convenience for the customer – we do not want customers to have to install any software in order to access our services. Also, we want them to be able to use any desktop machine, regardless of the operating system they have installed. The obvious choice for the client environment, therefore, is a web browser. Since the user interface must be interactive, in order to make a `Reservation` for example, we have to choose between technologies such as HTML/CGI, Java applets, ActiveX controls and Flash. Due to the need for portability (and client security), we can discount ActiveX controls. We would also prefer our customers to have almost-instant access to the user interface when browsing our site. This effectively discounts applets and Flash, both of which typically

involve a delay while the interface is downloaded. Therefore, the initial user interface will be HTML/CGI.

Once on the server side, servlets are a good choice for processing CGI requests because they're portable, they're efficient and they have access to all the facilities of J2EE, which provides everything that the servlets might need (such as access to distributed transaction management). Traditional scripting raises issues with portability, expressive power and performance; .Net technologies raise issues with portability. Once servlets are being used as the entry point to our servers, the obvious choice for producing dynamic web pages is the JSP mechanism.

Initially, we will use an open-source (free) implementation of J2EE for development and deployment, being careful to avoid any proprietary lock-ins. The implementation must support the forwarding of requests to servlets and JSPs running in a separate process, so that the latter can be accessed directly by GUI clients. Should the open-source implementations prove inadequate, we can simply purchase a commercial product and redeploy our code.

Because of their portability, we can deploy our servlets on any combination of hardware and operating system, and then redeploy at a later date if necessary. Initially, each store will have the system software deployed on two budget Linux servers, providing fail-over and throughput without the expense of a higher-powered server or highly-available hardware.

For the business data, we will use an open-source database initially, switching to a commercial product later if necessary. A relational database will be used because of the maturity of the technology and because our application is business-oriented, with a large quantity of data but no great logical complexity. The database will also be deployed on a pair of Linux servers at each store.

For the future, we aim to provide a user interface that can be used on a mobile phone or PDA. We would prefer to avoid WAP on devices that can't manage the HTML/CGI interface because it tends to be clumsy and unpopular. Instead, we will use J2ME, allowing us to provide a rich interactive experience that scales automatically to the size of the screen. For those customers who wish to install J2SE on their client, either manually or by using their Web browser's plug-in mechanism, we will also provide a conventional graphical user interface as an applet. This applet will be deployed on touch-screen kiosks in each store to enhance the customer experience, when viewing adverts for example.

It is anticipated that the J2ME and J2SE interfaces will bypass the JSPs and servlets, for improved performance.

## B.4.2 Layer Diagram

iCoot layers are illustrated in Figure B.27.

Persistence is provided by the JDBCLayer, using the standard JDBC library to access a relational database. There is no separate persistence layer, because we expect a relational database to serve our needs for the lifetime of the system.

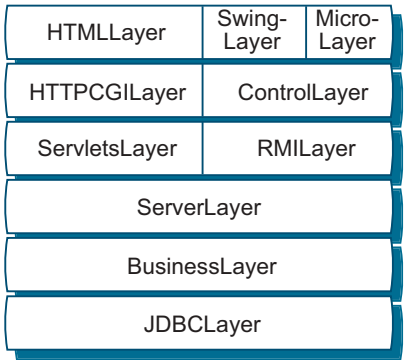


**Figure B.27: iCoot layer diagram**

The BusinessLayer contains implementations of the entity objects from the analysis class diagram, along with various supporting objects. These objects contain JDBC code for shipping data to and from the database.

The ServerLayer translates the objects and messages in the BusinessLayer into business services, in the form of messages on server objects. Objects in the ServerLayer are EJB session objects, which have two benefits: firstly, they give us access to J2EE transaction management; secondly, they allow us to provide direct access for GUI clients over RMI, bypassing the `WebServer`.

In order to keep the ServerLayer closed, all information returned by the business services takes the form of protocol objects, lightweight copies of the business objects.

The ServletsLayer is a control layer for HTML/CGI clients. Each servlet translates one or more objects in the ServerLayer into simple commands and questions that can be issued from the client. In response to each command or question, a servlet will perform whatever actions are necessary and then pass the next HTML page back to the client. So that page design and source code are separate, every reply page is built by a JSP that produces content dynamically based on the customer's interaction. The JSPs receive their dynamic data as protocol objects passed in by the servlets. The network communication for the HTMLLayer is provided by the standard HTTPCGILayer.

The RMILayer is a network layer allowing remote access from GUIs (Java applications and any device using J2ME). The objects in this layer are simply decorators for the EJB session objects in the ServerLayer: each server object is decorated with an RMI servant, while each RMI servant is accessed via an RMI proxy on the client. When communicating with the ControlLayer, the RMILayer uses the same protocol objects that the ServletsLayer uses when invoking JSPs.

The ControlLayer sits between the GUI objects and the RMI proxies. It serves to simplify interaction with the server objects and to hide the details of RMI. The RMILayer, ControlLayer, SwingLayer and MicroLayer are not documented fully because the graphical user interfaces are not part of the first increment of iCoot.

## B.4.3 Layer Interaction Policy

On the server, for the sake of simplicity, all layer communication will flow downwards. In other words messages will only be sent from one layer to the layer below. Events will be used on the client side for the benefit of the SwingLayer and the MicroLayer, so that application-specific knowledge can be pushed from the user interface components down to the ControlLayer. (The HTML/CGI front end does not need events because all information displayed to the user is calculated by the servlets and passed directly to JSPs for presentation.)

Layers will be closed, in order to make implementation and maintenance easier: each object will be able to access objects in the layer immediately below, but not beyond.

## B.4.4 Packages

The package diagram for Coot (including the graphical user interface/RMI packages not implemented in the first increment) is shown in Figure B.28.

## B.4.5 Deployment Diagram

The deployment diagram for iCoot is shown in Figure B.29.

The iCoot data tier comprises two database servers (which we have called DBServer). Having two such nodes improves throughput and reliability. Each DBServer hosts a DBMS process for managing access to data.

The cootschema.ddl artifact contains commands for creating database tables, in a format specific to the database being used. This is deployed to each DBMS process, using database-specific tools (no detail given here). Note that cootschema.ddl contains the schema for the full Coot system, since iCoot and Coot use the same data.

The middle tier, which communicates with the data tier, consists of two server machines (CootServer), again duplicated for the sake of reliability and throughput. Each CootServer hosts a CootBusinessServer (for handling business requests) and a WebServer (for handling static HTML content and forwarding business requests to the CootBusinessServer). Data access for the CootBusinessServer is provided by the DBMS. Because they're proprietary to the products that we select, the communication protocols between the WebServer

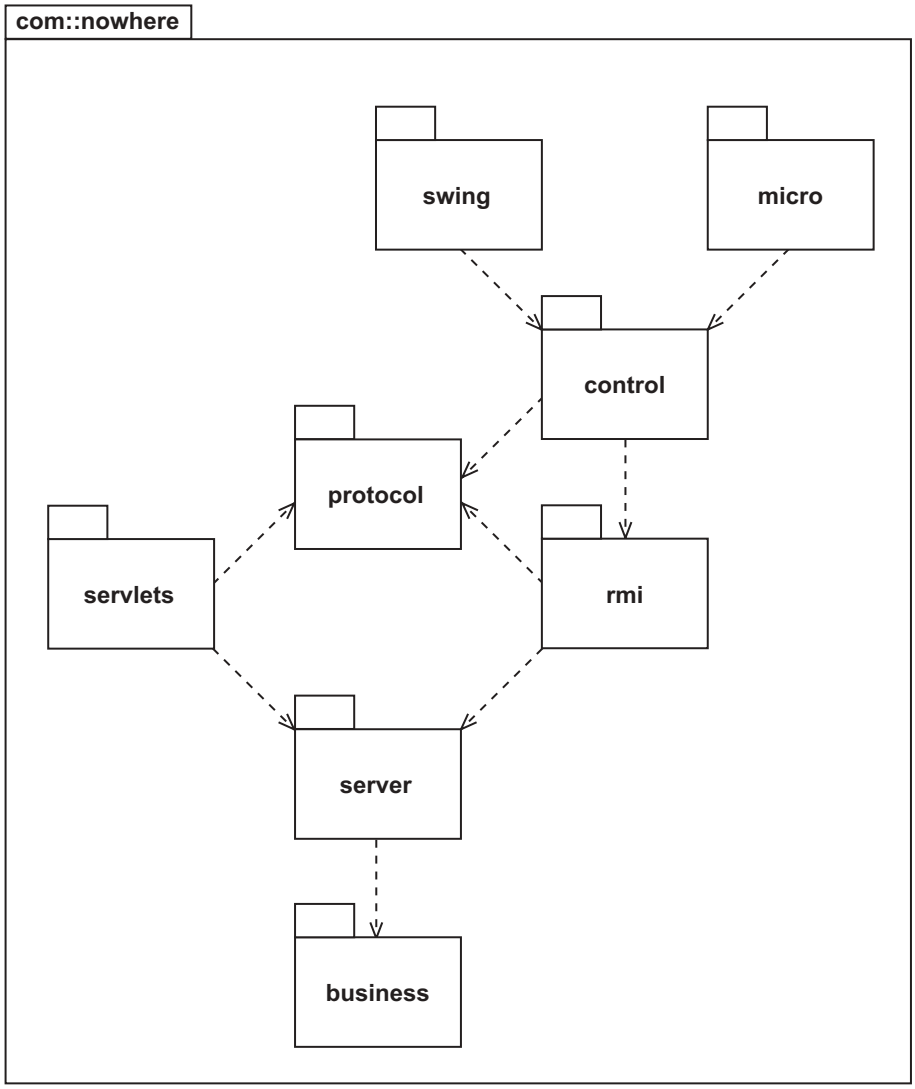


**Figure B.28: iCoot package diagram**

and the CootBusinessServer and between the CootBusinessServer and the DBMS are not specified.

Within each CootServer, the iCoot folder, containing static HTML pages, is deployed to the WebServer, while the icoot.ear archive is deployed to the CootBusinessServer. The icoot.ear archive contains servlets, JSPs, business objects and (eventually) RMI decorators, from the com::nowhere package.

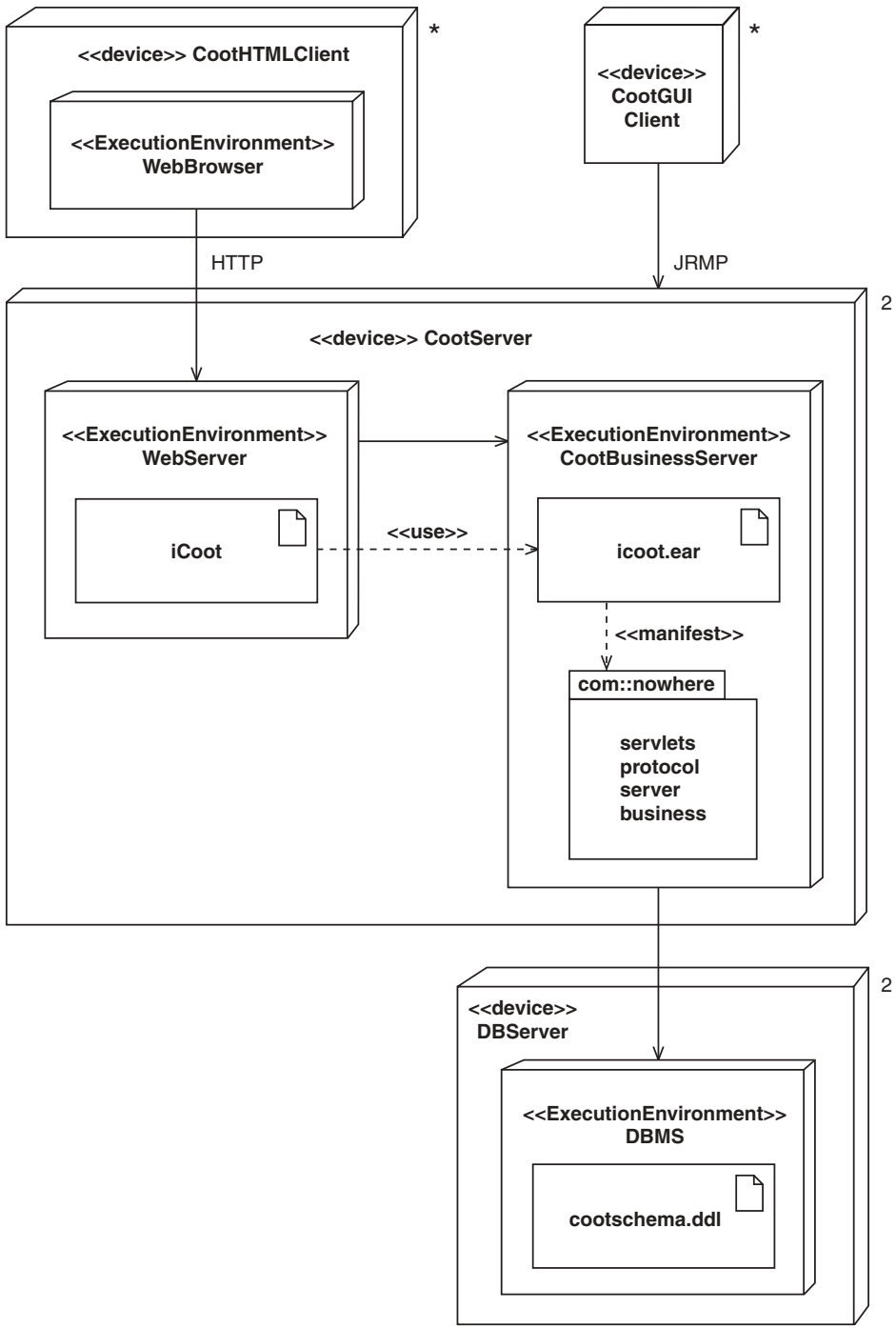


**Figure B.29: iCoot deployment diagram**

Each CootServer can be accessed simultaneously by any number of CootHTMLClients. Each CootHTMLClient hosts a WebBrowser, which accesses one of the WebServers using HTTP. No artifacts need to be deployed to the CootHTMLClients.

Eventually, we will also provide access from CootGUIClients. Each CootGUIClient will access one of the CootServers, using JRMP. Because the mechanism that allows such requests to get into the CootBusinessServer is the subject of a future increment, no details are given. Nor is any detail given for the CootGUIClient processes. The artifacts deployed to the CootGUIClients, if any, are not specified.

## B.4.6 Security Policy

Searching and browsing services will be available to all-comers, without logging in. In contrast, for Member-only services, each Member must first obtain a password in person from their local Store, then use it to log in to the Member's area from their chosen client. The membership numbers and passwords used to log in will be managed in a central directory, for ease of maintenance, using the standard Java integration mechanisms.

In order to keep Member activity private, all access to Member services from the client will be over SSL rather than plain TCP/IP. SSL will also be employed between servers for internal protection.

The servers will be deployed behind an Internet firewall so that external access can be tightly controlled.

## B.4.7 Concurrency Policy

Objects in the BusinessLayer will be managed using distributed transactions. At the start of each business request (i.e. each method on a ServerLayer object), a Java transaction will be created – this transaction will be associated with every database access made by business objects within that request. At the end of each request, the Java transaction will be committed, thus making updates available to other requests.

In order to minimize transaction conflict, all RMI servants, servlets and server objects will be stateless.

For GUI clients, access to local data (copies of protocol objects) will be single-threaded. For HTML clients, each JSP will have exclusive access to its protocol objects, so such access is also effectively single-threaded. For business data, low-level concurrency control is managed automatically by the EJB framework: every use of a business service is wrapped inside a transaction, which passes right through to the database management system.

To simplify concurrency control at the business level, two strategies will be employed: firstly, single log-on will be enforced for Members. Secondly, updates to the Catalog of available CarModels will be made off-line and switched with the live Catalog in the early hours of the morning. This will minimize the need to report errors such as 'Attempt to reserve a car model that has been withdrawn' to Customers and Assistants. (This will still occasionally happen, because client displays won't be updated explicitly when data changes on the server via concurrent paths; pushing all relevant changes to clients, although technically feasible, would be too inefficient.)

# B.5 SUBSYSTEM DESIGN

This section documents the results of subsystem design, carried out during the design phase of the iCoot development, in terms of business services, design class models (one per layer plus the protocol objects), database schema, user interface design and business service realization (sequence diagrams). In general, this documentation describes the subsystem artifacts required to support business service realization and the CootHTMLClient.

## B.5.1 Business Services

1. Read headings from the CarModel index.
2. Read CarModels for a given index heading.
3. Read all CarModel Categories.
4. Read all CarModel engine sizes.
5. Read all Makes of CarModel.
6. Read CarModels for a given set of Categories, engine sizes and Makes.
7. Read details for a given CarModel.
8. Reserve a CarModel.
9. Read details for a given Member.
10. Change a Member's password.
11. Read the Cars rented by a given Member.
12. Read the Reservations made by a given Member.
13. Cancel a Reservation.

## B.5.2 ServletsLayer Class Diagram

The class diagram for the ServletsLayer is shown in Figure B.30.

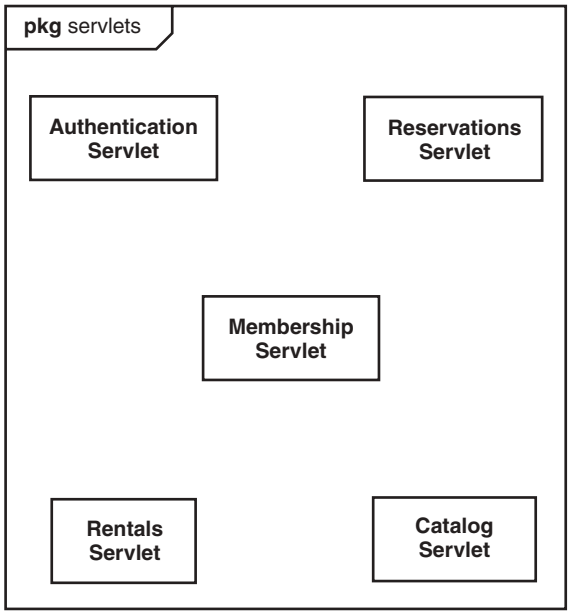


**Figure B.30: ServletsLayer classes**

## B.5.3 ServletsLayer Field List

All servlets are stateless and therefore do not have any fields. Any information about the state of a customer's interaction is recorded in the result pages themselves and in a state object stored in the browser's HTTP session. For logged-on members, the session identifier is also stored in the HTTP session. Objects in the server layer are located using their homes (implemented using the Singleton pattern).

## B.5.4 ServletsLayer Message List

Using the standard Java mechanism, each client request is passed to the selected servlet via a message called doGet(:HttpServletRequest,:HttpServletResponse). The questions and commands from the client are passed in as part of the HttpServletRequest. Below is a list of the questions and commands that can be passed in by the client, along with the JSPs that are invoked as a result.

AuthenticationServlet

- logon From home page; takes a membership number, password and 'steal' parameter and, upon successful logon, returns the member page. (The steal parameter indicates whether the member would like to steal an existing session.)
- logoff From member page; logs off the current Member.

CatalogServlet

- index From member or non-member page; returns the index page containing index headings to choose from.
- browse From index page; takes an index heading as parameter and returns the results page containing matching CarModels.
- search From member or non-member page; returns the search page containing every Category, Make and engine size to choose from.
- query From search page; takes Category ids, Make ids and engine sizes as parameters and returns the results page containing matching CarModels.
- details From results page; takes a CarModel id as parameter and returns the details page containing details for that CarModel.

MembershipServlet

- membership From member page; returns the membership page, containing details for the current Member.
- password From member page; returns the password page.
- changePassword From member page; takes an old password and a new password as parameters and returns the confirmChange page.
- confirmChange From the confirmChange page; sets the new password for the current Member, if the old password given was correct.

RentalsServlet

- rentals From member page; returns the rentals page, containing Rentals for the current Member.

ReservationsServlet

- reserve From details page; takes a CarModel id as parameter and returns the confirmReserve page.
- confirmReserve From confirmReserve page; reserves the CarModel already identified and returns the confirmation page.
- ok From confirmation page; returns the details page.
- reservations From member page; returns the reservations page, containing Reservations for the current Member.
- cancel From the reservations page; takes the Reservation id as parameter and returns the confirmCancel page.
- confirmCancel From confirmCancel page; cancels the previously selected Reservation and returns the reservations page.

## B.5.5 ServerLayer Class Diagram

The class diagram for the ServerLayer is shown in Figure B.31. Each class also has a home, implemented using the Singleton pattern (not shown).

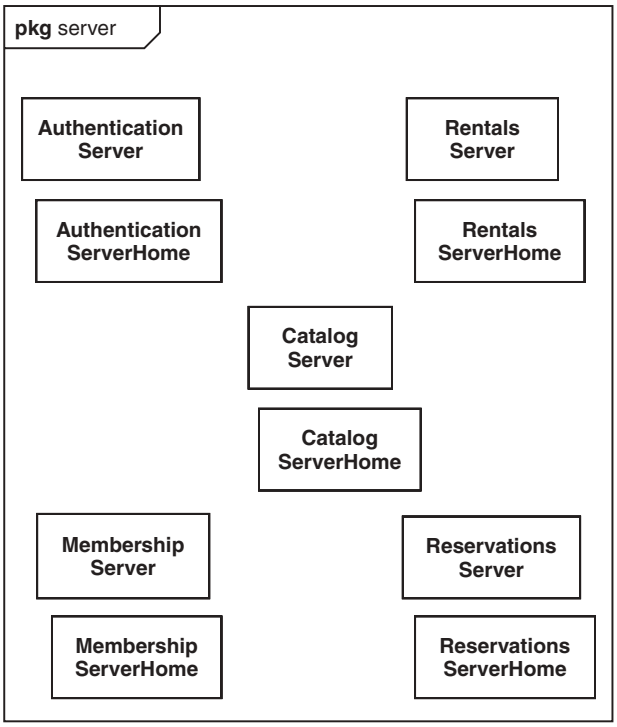


**Figure B.31: ServerLayer classes**

## B.5.6 ServerLayer Field List

The server objects are stateless, thus they have no fields. (All BusinessLayer classes are accessed via their respective homes.)

## B.5.7 ServerLayer Message List

For the server objects, the messages below correspond to the business services. (Homes have been omitted from this list because they simply create server objects, with no parameters.)

AuthenticationServer

- +logon(n:String,p:String,s:boolean):long  Log on the Member with membership number n and password p, specifying whether or not to steal any existing session with s.
- +logoff(i:int)  Log off the Member with session identifier i.

CatalogServer

- +readCategoryNames():String[] Read the names of every Category.
- +readMakeNames():String[] Read the names of all Makes.
- +readEngineSizes():int[] Read the unique engine sizes of all CarModel details.
- +readIndexHeadings():String[] Read the index headings, derived from all CarModel numbers and Make names.
- +readCarModels(h:String):PCarModel[] Read all CarModels matching the index heading h.
- +readCarModelDetails(i:int):PCarModelDetails Read details for the CarModel with identifier i.
- +readCarModels(q:PCatalogQuery):PCarModel[] Read all CarModels that match the query q.

MembershipServer

- +readMember(i:int):PMember Read the Member with session identifier i.
- +changePassword(i:int,o:String,n:String) Change the password for the Member with session identifier i, using old password o and new password n.

RentalsServer

- +readRentals(i:int):PRental[] Read all Rentals for the Member with session identifier i.

ReservationsServer

- +readReservations(i:int):PReservation[] Read all Reservations for the Member with session identifier i.
- +createReservation(i:int,c:int) Create a Reservation for the Member with session identifier i and the CarModel with identifier c.
- +cancelReservation(i:int,r:int) Cancel the Reservation with identifier r, for the Member with session identifier i, as long as the Reservation matches the Member.

## B.5.8 BusinessLayer Class Diagram

The class diagram for the BusinessLayer is shown in Figure B.32. Most of these classes also appear in the analysis class model (Section B.3), so their descriptions have been placed in the Glossary (Section B.8), to avoid repetition.

Each entity class has a home, apart from Customer (because it's abstract). ReservationStateHome is an Abstract Factory for creating instances of its subclasses.

The Store class, used for illustration in Chapter 13, has not been included in this appendix since it plays no part in business service realization.

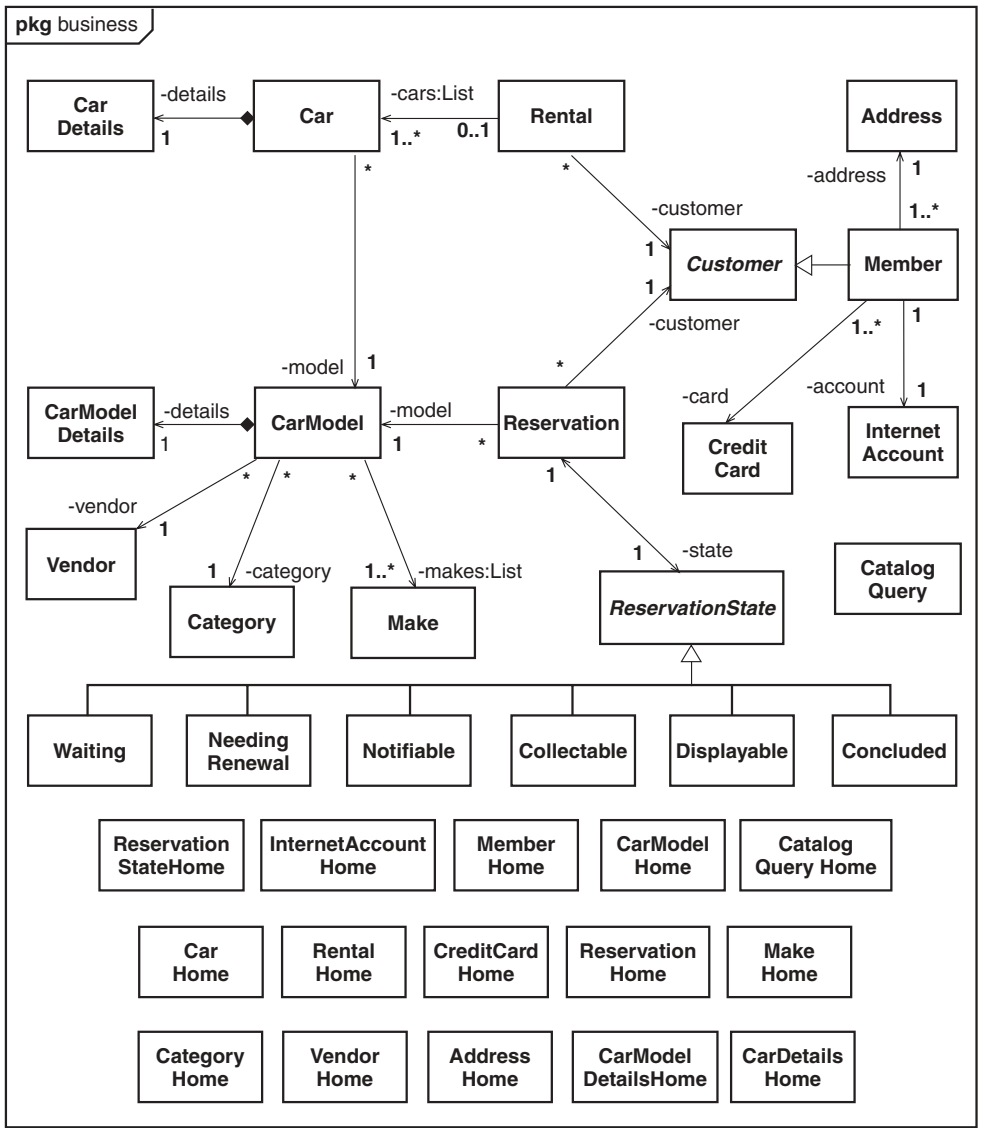


**Figure B.32: BusinessLayer classes**

## B.5.9 BusinessLayer Field List

The list below shows fields for stored attributes only; fields that store links are shown in Figure B.32. With the exception of the state objects, every object has an -id:int that stores the universal identifier but which has been omitted from the list. Classes with no fields other than links have been omitted.

Address

- -house:String House number and/or name (unique within the postal code).
- -street:String Street in which the house stands.
- -county:String County where the street is.
- -postCode:String Postal code for the sorting office and region.

Car

- -traveled:int The distance, in kilometers, that the Car has traveled (taken from the odometer).
- -dateLost:Date Date the Car was reported missing, null if not lost.

CarDetails

- -barCode:String The bar code, attached inside the Car's windscreen.
- -numberPlate:String The Car's license number as it appears on the plate.
- -vin:String The Car's unique Vehicle Identification Number, on a plate riveted to the body.

CarModel

- -name:String The Make's unique name for the CarModel.
- -price:int The daily cost of hiring the Car, in cents.

CarModelDetails

- -engineSize:int The capacity of the engine, in cubic centimeters.
- -description:String A single-sentence description of the CarModel.
- -advert:String The file name of a streaming advert for the CarModel.
- -poster:String The file name of a poster for the CarModel.

CatalogQuery

- -makeIds:List<Integer> Universal identifiers for relevant Makes (using identifiers avoids retrieving Makes when building the database query).
- -carModelIds:List<Integer> Universal identifiers for relevant CarModels (using identifiers avoids retrieving CarModels).
- -engineSizes:List<Integer> Engine sizes, in cubic centimeters.

Category

- -name:String The name of the Category.

Collectable

- -dateNotified:Date Date the Customer was notified that the car is ready for collection.

Concluded

- -reason:String Why the Reservation is concluded: "Successfully rented", "Canceled by customer", "Wasn't renewed" or "Wasn't collected".

CreditCard

- -type:String Type of Card (e.g. "Annex").
- -number:String Number on the Card.
- -expiryDate:Date Date Card expires.

Customer

- -name:String The Customer's name.
- -phone:String The Customer's phone number.
- -amountDue:int Fees due, in cents, for the Customer (for example, for overdue Rentals).

Displayable

- -reason:String Why the Car must be put back in the display area, either "Customer uncontactable" or "Failed to collect".

InternetAccount

- -password:String Password for the associated Member; must be a mix of at least six letters and digits.
- -sessionId:long Session identifier for the associated Member: 0 if not logged on; securely random, unique, non-zero number if logged on.

Make

- -name:String The name of the manufacturer.

Member

- -number:String The Member's unique number.
- -inGoodStanding:boolean Whether or not the Member has any outstanding issues, such as disputed late fees.

NeedingRenewal

- -dateRenewalNeeded:Date Date by which the Reservation must be renewed to avoid becoming automatically concluded.

Notifiable

- -datePutAside:Date Date the Car was moved to the reserved area.

Rental

- -number:String The unique number for the Rental.
- -startDate:Date The date the Rental was taken out.
- -dueDate:Date The date the Rental is due to end.
- -totalAmount:int The amount paid for the Rental, in cents.

Reservation

- -number:String The unique number for the Reservation.
- -timestamp:Timestamp The date and time that the Reservation was made.

Vendor

- -name:String Name of the Vendor.

Waiting

- -lastRenewedDate:Date Date the Reservation was last renewed (initially the same as the date it was created).

## BusinessLayer Message List

Most of the public messages on the BusinessLayer classes are simply accessors for attributes, derived attributes or links, so no detail is given for these.

For the ReservationState hierarchy, and the Reservation class itself, every class has one message for each event shown in Figure B.14 and one getter for each state attribute. Details of these messages have been omitted, for brevity. In addition, the Reservation class has test messages to enable clients to discover which state the receiver is in, e.g. isConcluded and isWaiting.

For the homes, details resulting from the Singleton pattern are not given. To support the homes, every class created by a home has a package constructor that takes all attributes and links as parameters; no further detail is given for these constructors.

Apart from ReservationStateHome and CatalogQueryHome, every home has the following messages (where X is a stand-in for the corresponding BusinessLayer class):

- +findByPrimaryKey(id:int):X Returns the instance of X with universal identifier id.
- +create(...):X Takes every attribute and link as parameters and returns a new instance of X.

ReservationStateHome has one creation message for each subclass, taking all subclass attributes as parameters.

CatalogQueryHome has a create message that takes three List<Integer> parameters: makeIds, categoryIds and engineSizes.

All other home messages used in business service realization are listed below.

CarModelHome

- +findByIndexHeading(h:String):List<CarModel> Returns all CarModels that have h as a model number or Make name, sorted by model name.
- +findByQuery(q:CatalogQuery):List<CarModel> Returns all CarModels that have a Make, a Category and an engine size that appear in q, sorted by model name.

CarModelDetailsHome

- +findByCarModelId(id:int):CarModelDetails Returns the instance that matches the CarModel with universal identifier id.
- +findEngineSizes():List<Integer> Returns all engine sizes, in ascending order.

CategoryHome

- +findCategoryNames():List<String> Returns all Category names, in alphabetical order.

MakeHome

- +findMakeNames():List<String[]> Returns all Make names, in alphabetical order.

MemberHome

- +findByMembershipNumber(n:String):Member Returns the Member with membership number n.
- +findBySessionId(id:long):Member Returns the Member with session identifier id.

RentalHome

- +findByMember(m:Member):List<Rental> Returns all Rentals for m, sorted by start date.

ReservationHome

- +findUnconcludedByMember(m:Member):List<Rental> Returns all the unconcluded Reservations for m, sorted by creation date.

## B.5.10 Protocol Objects Class Diagram

The class diagram for the protocol objects, used for communication between the server layer and the servlets layer, is shown in Figure B.33.

### Protocol Objects Field List

The list below shows fields for stored attributes only; fields that store links are shown in Figure B.33. The meaning of these fields is the same as those for the BusinessLayer classes, with the exception that cardNumber only includes the last four real digits. The message variable on PServerException stores an explanation for the exception.

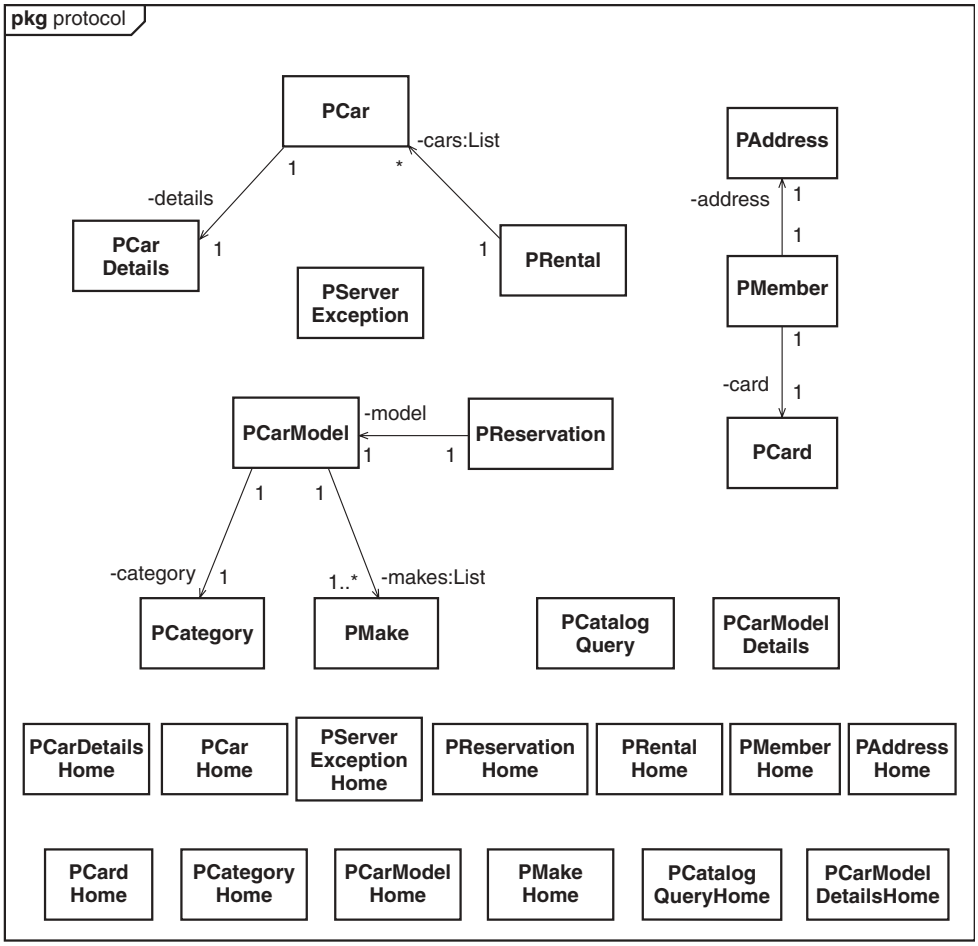


**Figure B.33: Protocol classes**

- PAddress -house:String, -street:String, -county:String, -postCode:String
- PCar -traveled:int, -numberPlate:String
- PCarModel -id:int, -name:String, -price:int
- PCarModelDetails -engineSize:int, -description:String, -advert:String, -poster:String
- PCatalogQuery -makeIds:int[], -carModelIds:int[], -engineSizes:int[]
- PCategory -name:String
- PCreditCard -type:String, -number:String
- PMake -name:String
- PMember -name:String, -phone:String, -amountDue:int, -inGoodStanding:boolean
- PRental -startDate:Date, -dueDate:Date

- PReservation -id:int, -number:String, -timestamp:Timestamp
- PServerException -message:String

### Protocol Objects Message List

Nearly all the messages on the protocol classes are accessors for attributes or links. In addition, each protocol class has a toString message that returns a human-readable summary of the receiver: these messages are used to display objects in user interfaces.

Each home has a single create message with a parameter to initialize each attribute and link. Beyond this, all messages on homes are derived from the Singleton pattern. To support the homes, each protocol class has a package constructor that takes every attribute and link as parameters. No further detail is given for these constructors.

In addition to its basic creation message, PCatalogQueryHome has a createCatalogQuery message that takes three int[] parameters (makeIds, categoryIds and engineSizes) and returns a new CatalogQuery.

Since none of the protocol classes or homes have any special messages, no detailed list is given here.

## B.5.11 Database Schema

The database schema is shown in Figure B.34. In this diagram, the names of primary key columns are shown in bold and the names of foreign key columns are shown in italics. The meaning of the attribute columns is the same as those for the fields in the BusinessLayer classes. Of all the columns, only DATELOST in the CAR table is nullable.

## B.5.12 User Interface Design

The user interface design is not given here since it is similar to the user interface sketches shown in Section B.3.

## B.5.13 Business Service Realization

Figures B.35 through B.46 show one sequence diagram, documenting business service realization, per system use case. Several diagrams use frames to show loops. The loop operator has a pseudocode guard controlling the number of iterations. Frames have also been used to enclose lifelines that refer to included use cases, using the ref operator – in this case, the name of the included use appears in the center of the frame, alongside a list of the parameters that are passed in.

For the sake of simplicity, each interaction between a customer's browser and a servlet has been shown as a message sent from the actor directly to the servlet. In reality, this is implemented by passing a command and its parameters into the servlet's doGet message.

**ADDRESS** (**ID**:INTEGER,HOUSE:VARCHAR(99),STREET:VARCHAR(99),COUNTY:VARCHAR(99),
POSTCODE:VARCHAR(99))

**CAR** (**ID**:INTEGER,TRAVELLED:INTEGER,DATELOST:DATE,*CARDETAILSID*:INTEGER)

**CARD** (**ID**:INTEGER,TYPE:VARCHAR(99),NUMBER:VARCHAR(99))

**CARDETAILS** (**ID**:INTEGER,BARCODE:VARCHAR(99),NUMBERPLATE:VARCHAR(99),VIN:VARCHAR(99))

**CARMODEL** (**ID**:INTEGER,NAME:VARCHAR(99),PRICE:INTEGER,*CARMODELDETAILSID*:INTEGER,
*CATEGORYID*:INTEGER,VENDORID:INTEGER)

**CARMODELDETAILS** (**ID**:INTEGER,ENGINESIZE:VARCHAR(99),DESCRIPTION:VARCHAR(256),
ADVERT:VARCHAR(99),POSTER:VARCHAR(99))

**CATEGORY** (**ID**:INTEGER,NAME:VARCHAR(99))

**COLLECTABLERESERVATION** (***RESERVATIONID***:INTEGER,DATENOTIFIED:DATE)

**CONCLUDEDRESERVATION** (***RESERVATIONID***:INTEGER,REASON:VARCHAR(99))

**CUSTOMER** (**ID**:INTEGER,NAME:VARCHAR(99),PHONE:VARCHAR(99),AMOUNTDUE:INTEGER)

**DISPLAYABLERESERVATION** (***RESERVATIONID***:INTEGER,REASON:VARCHAR(99))

**INTERNETACCOUNT** (**ID**:INTEGER,PASSWORD:VARCHAR(99),SESSIONID:INTEGER)

**MAKE** (**ID**:INTEGER,NAME:VARCHAR(99))

**MAKECARMODEL** (***CARMODELID***:INTEGER,***MAKEID***:INTEGER)

**MEMBER** (**ID**:INTEGER,NUMBER:VARCHAR(99),INGOODSTANDING:BOOLEAN,*CARDID*:INTEGER,
*ADDRESSID*:INTEGER)

**NEEDINGRENEWALRESERVATION** (***RESERVATIONID***:INTEGER,DATERENEWALNEEDED:DATE)

**NONMEMBER** (**ID**:INTEGER,DRIVERSLICENSE:VARCHAR(99))

**NOTIFIABLERESERVATION** (***RESERVATIONID***:INTEGER,***DATEPUTASIDE***:DATE)

**RENTAL** (**ID**:INTEGER,NUMBER:VARCHAR(99),STARTDATE:DATE,DUEDATE:DATE,TOTALAMOUNT:INTEGER)

**RENTALCAR** (***RENTALID***:INTEGER,***CARID***:INTEGER)

**RESERVATION** (**ID**:INTEGER,NUMBER:VARCHAR(99),TIMESTAMP:TIMESTAMP,*CUSTOMERID*:INTEGER,
*CARMODELID*:INTEGER)

**VENDOR** (**ID**:INTEGER,NAME:VARCHAR(99))

**WAITINGRESERVATION** (***RESERVATIONID***:INTEGER,LASTRENEWEDDDATE:DATE)

**Figure B.34: Database schema**

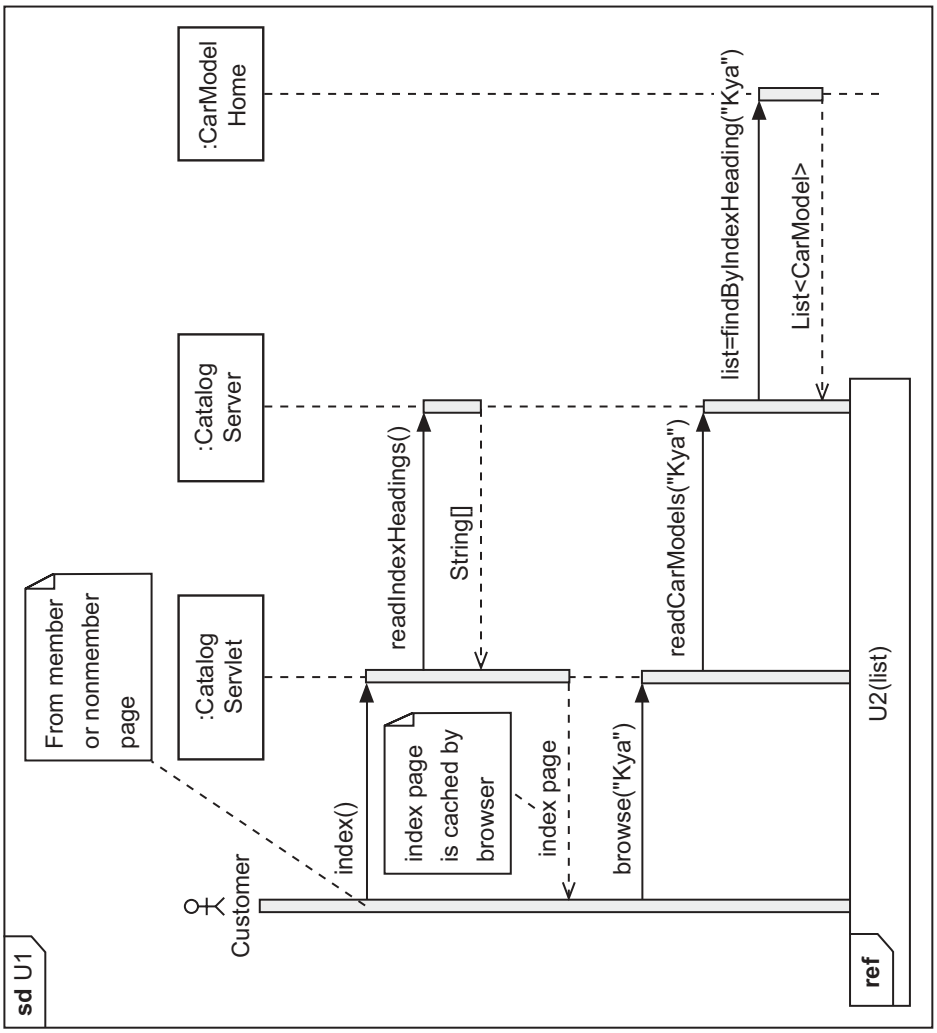


**Figure B.35: Sequence diagram for U1:Browse Index**

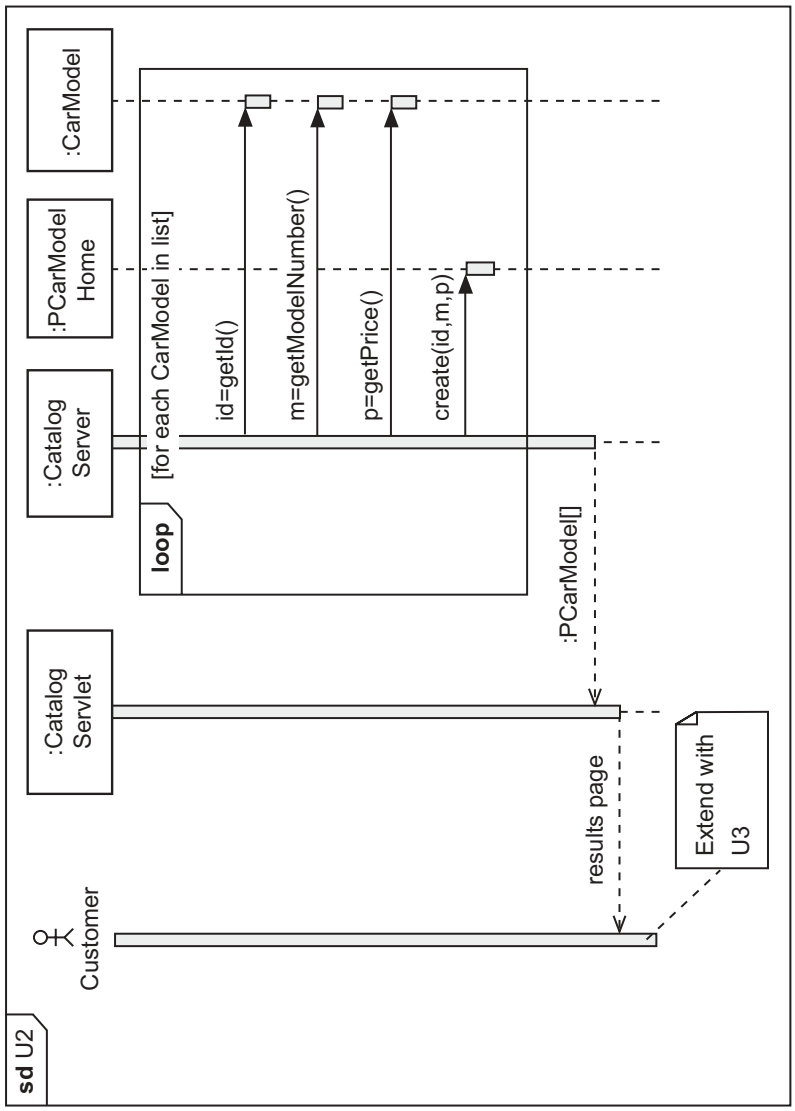


**Figure B.36: Sequence diagram for U2:View Results**

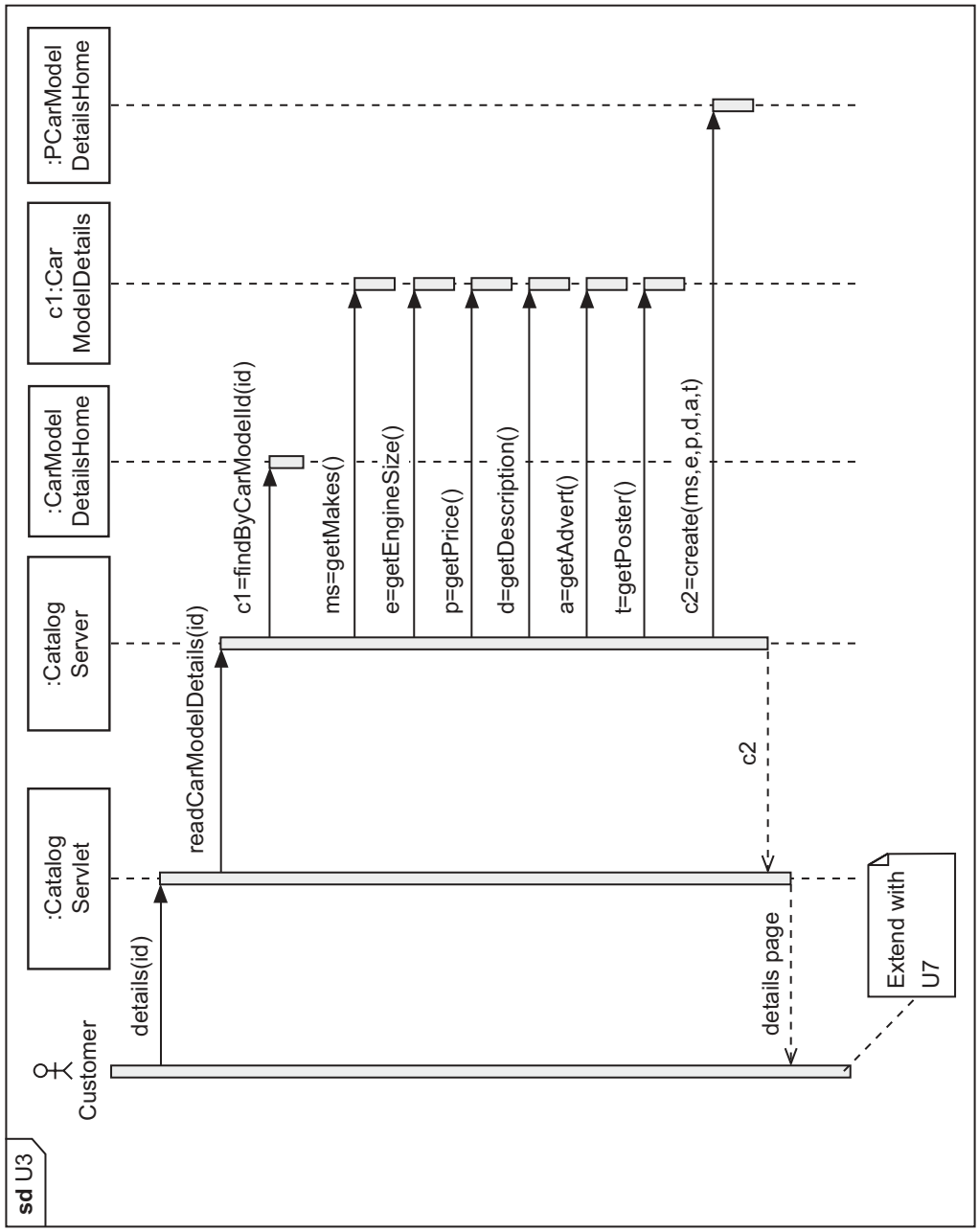


**Figure B.37: Sequence Diagram for U3:View CarModel Details**

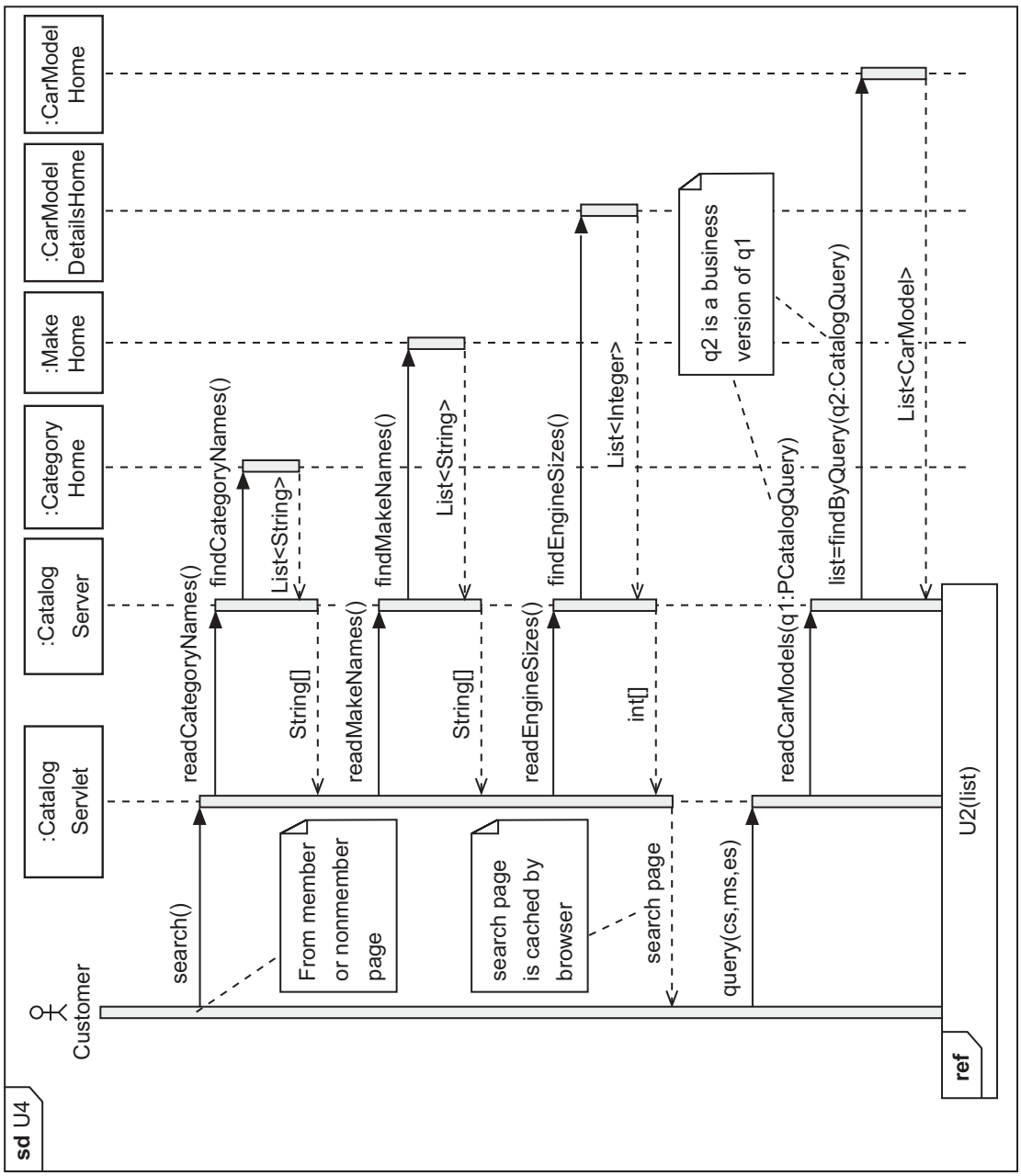


**Figure B.38: Sequence diagram for U4:Search**

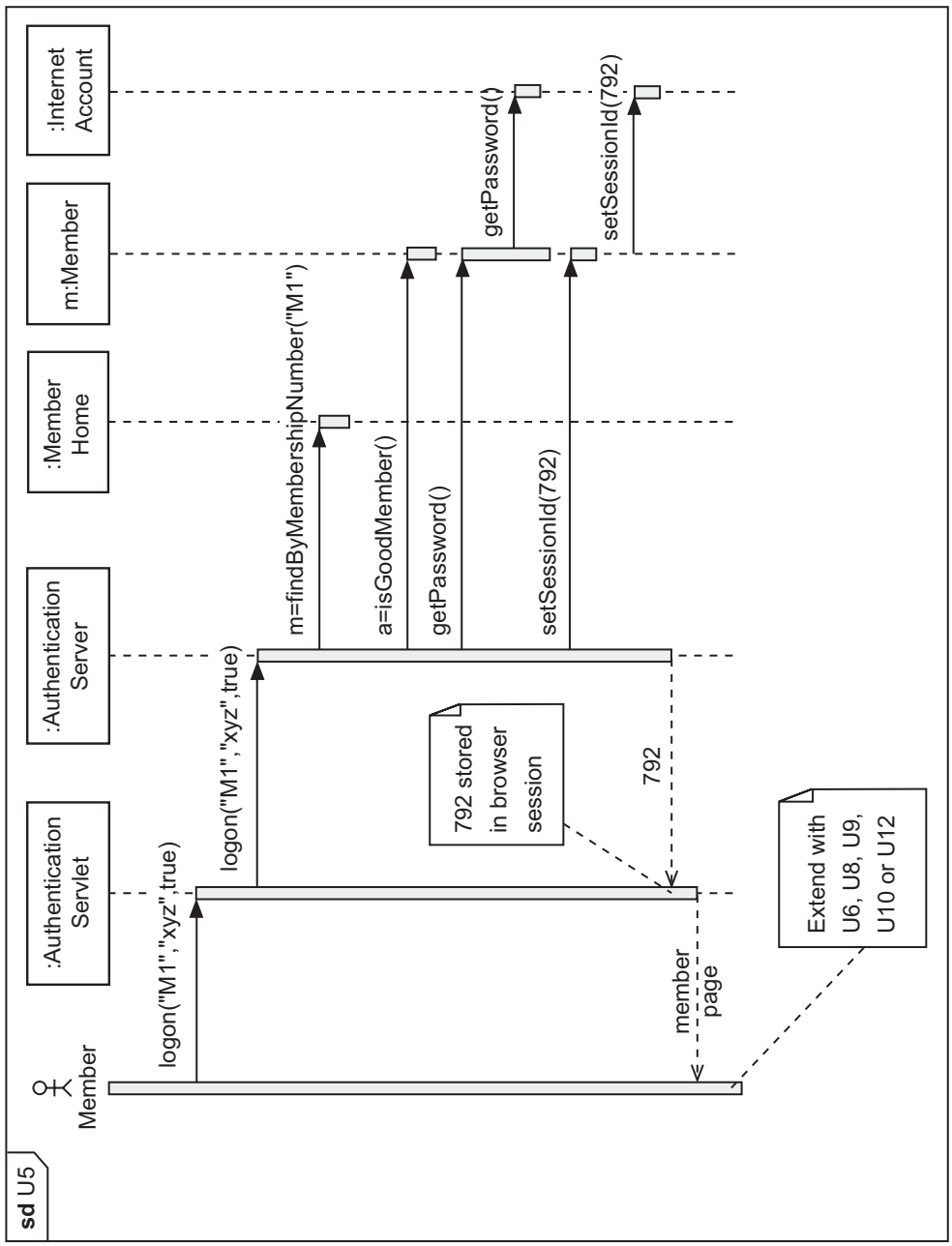


**Figure B.39: Sequence diagram for U5:Log On**

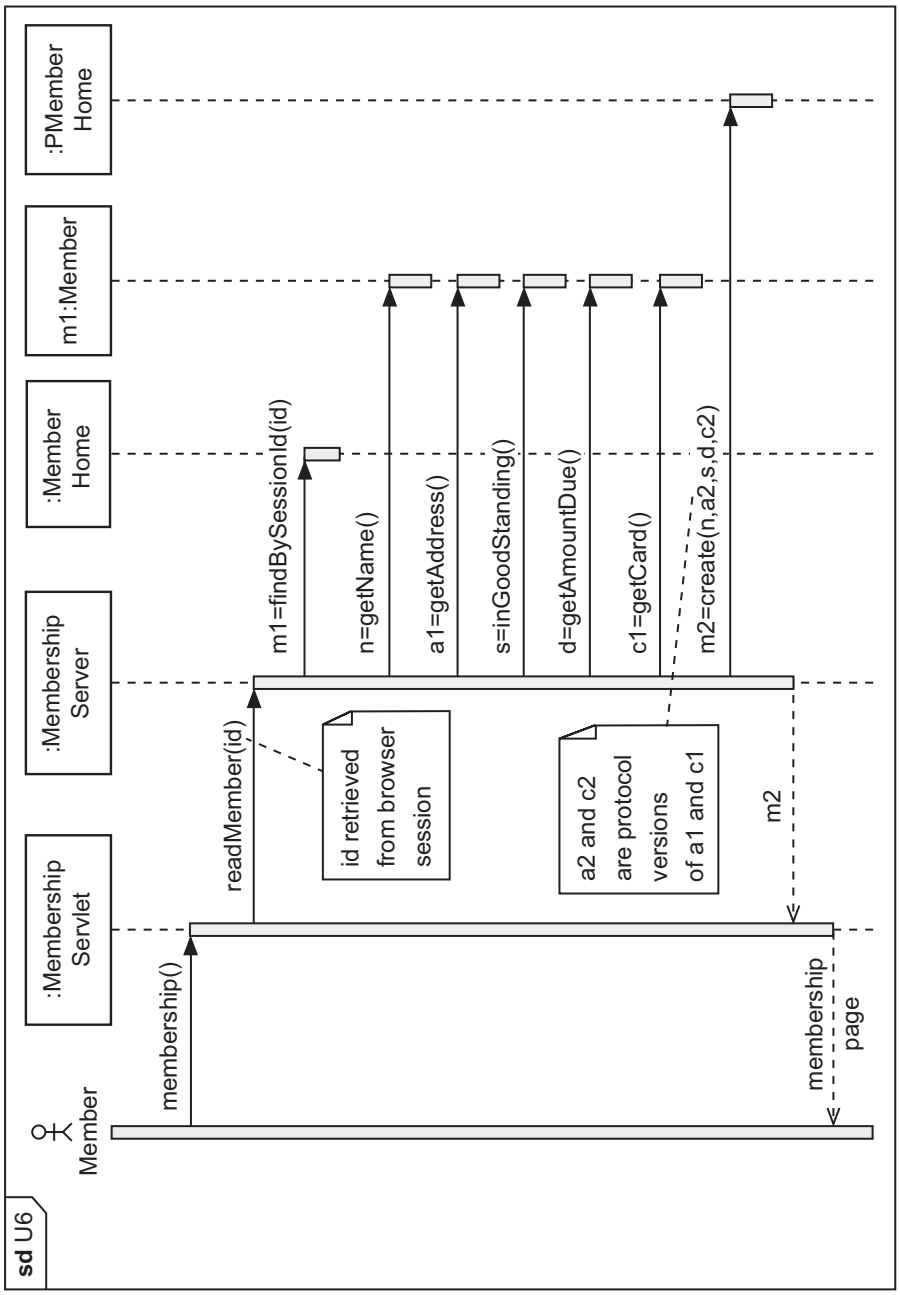


**Figure B.40: Sequence diagram for U6:View Member Details**

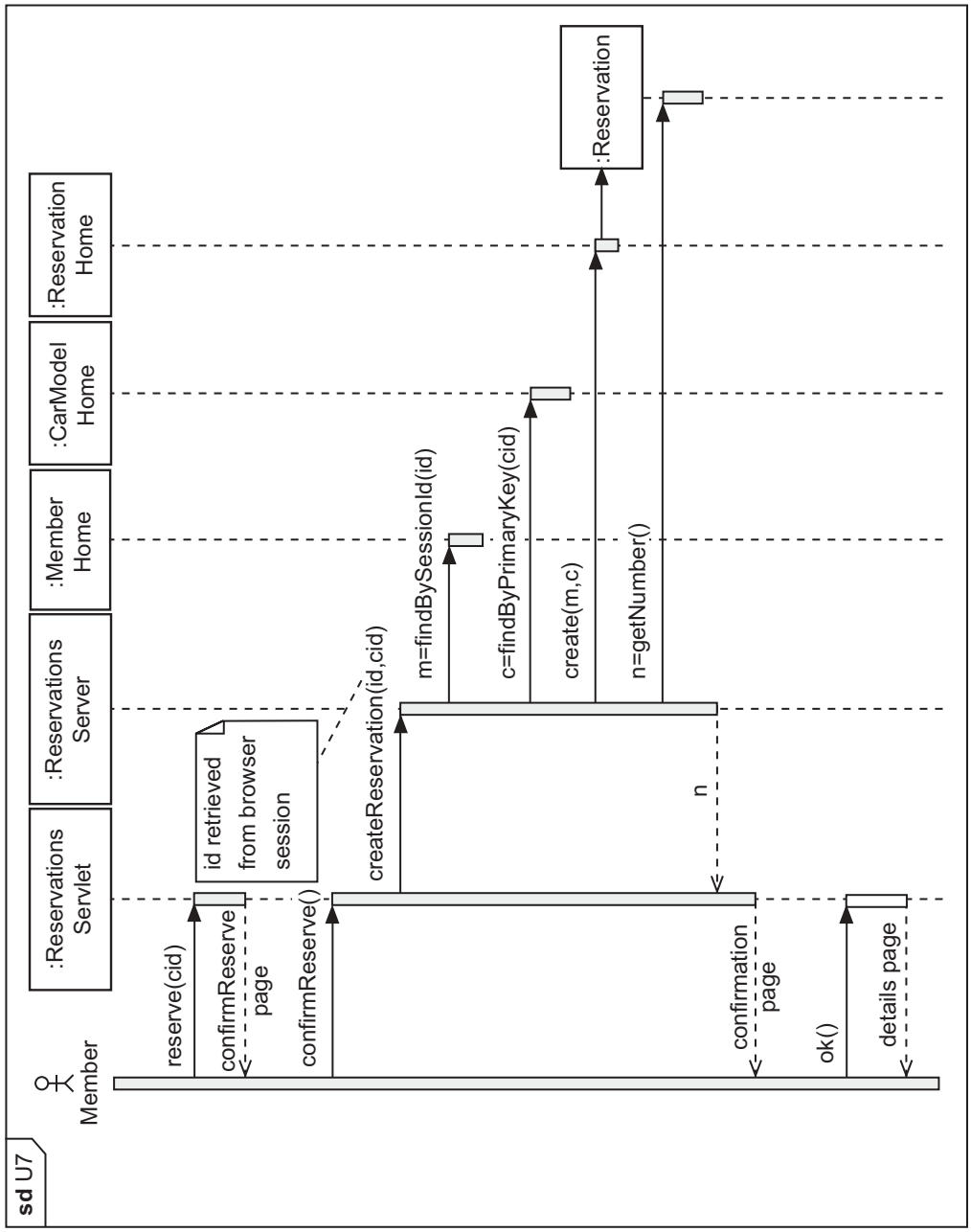


**Figure B.41: Sequence diagram for U7:Make Reservation**

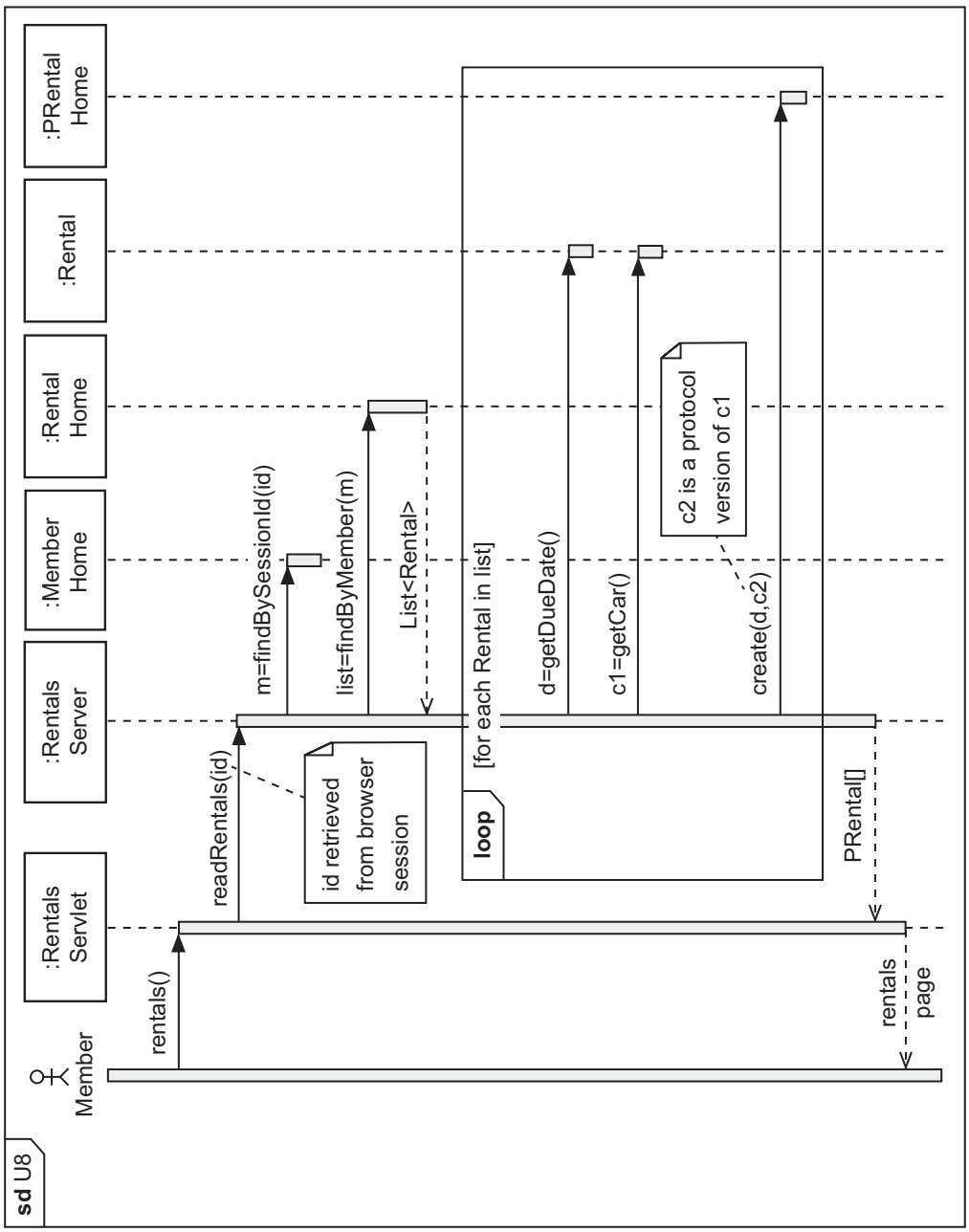


**Figure B.42: Sequence diagram for U8:View Rentals**

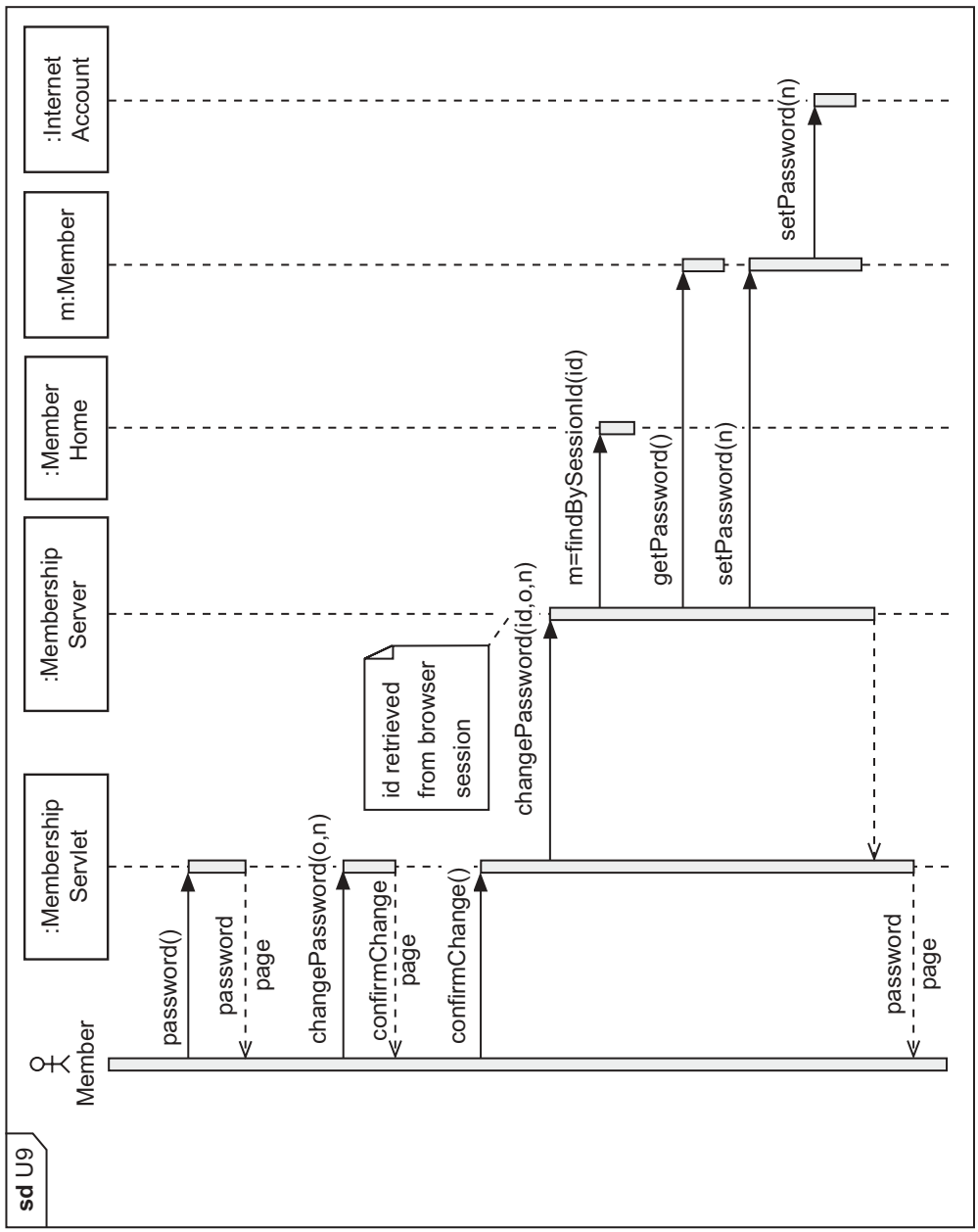


**Figure B.43: Sequence diagram for U9:Change Password**

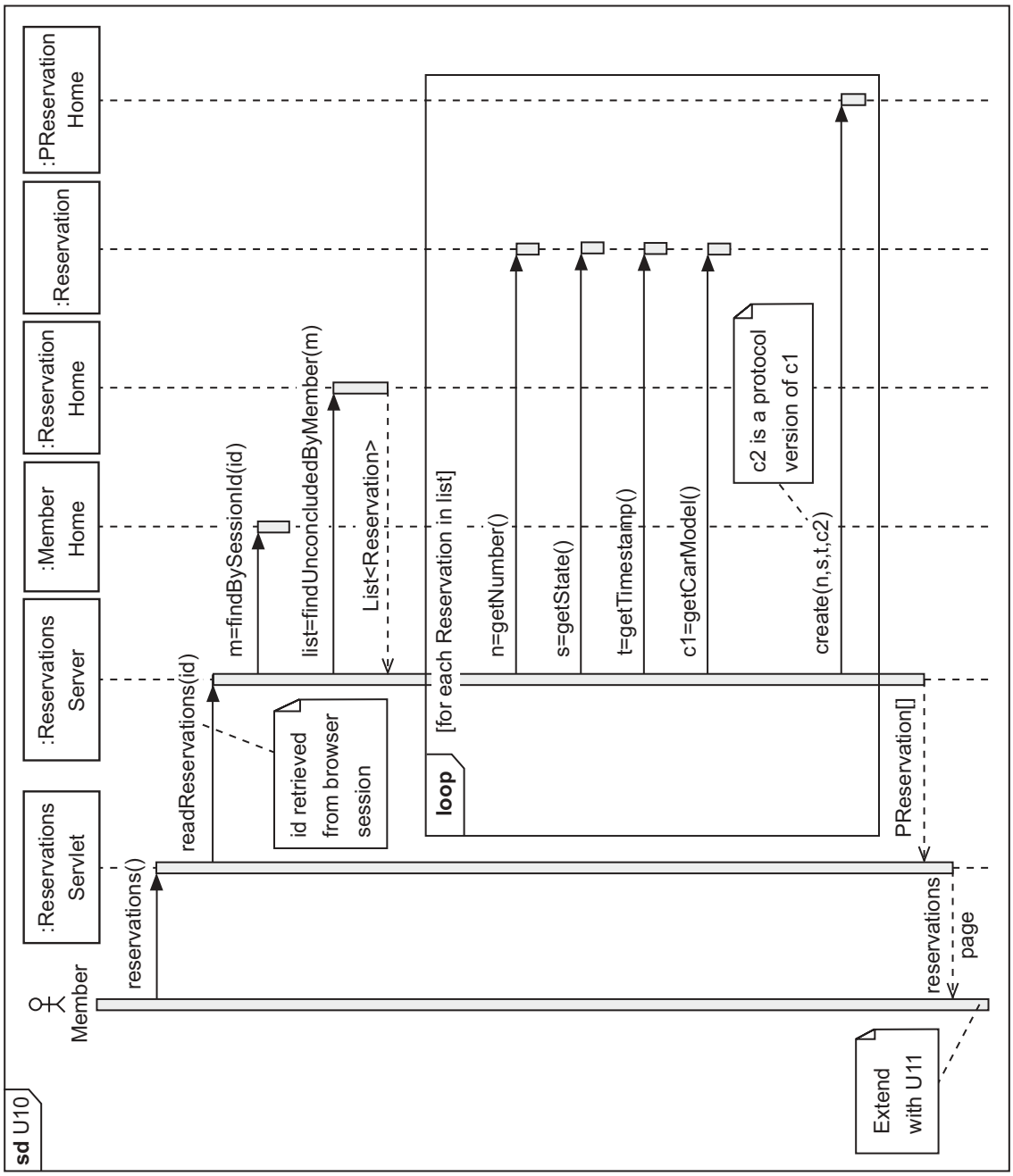


**Figure B.44: Sequence diagram for U10: View Reservations**

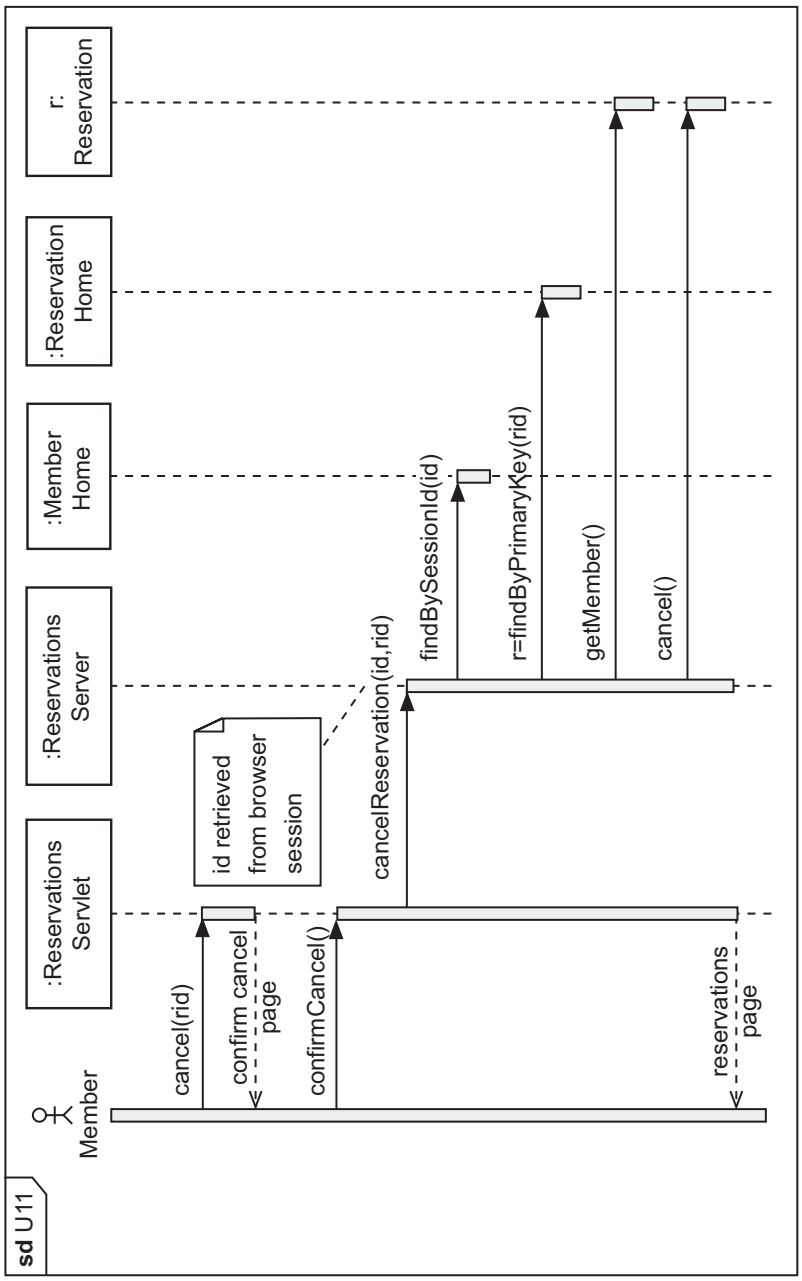


**Figure B.45: Sequence diagram for U11:Cancel Reservation**

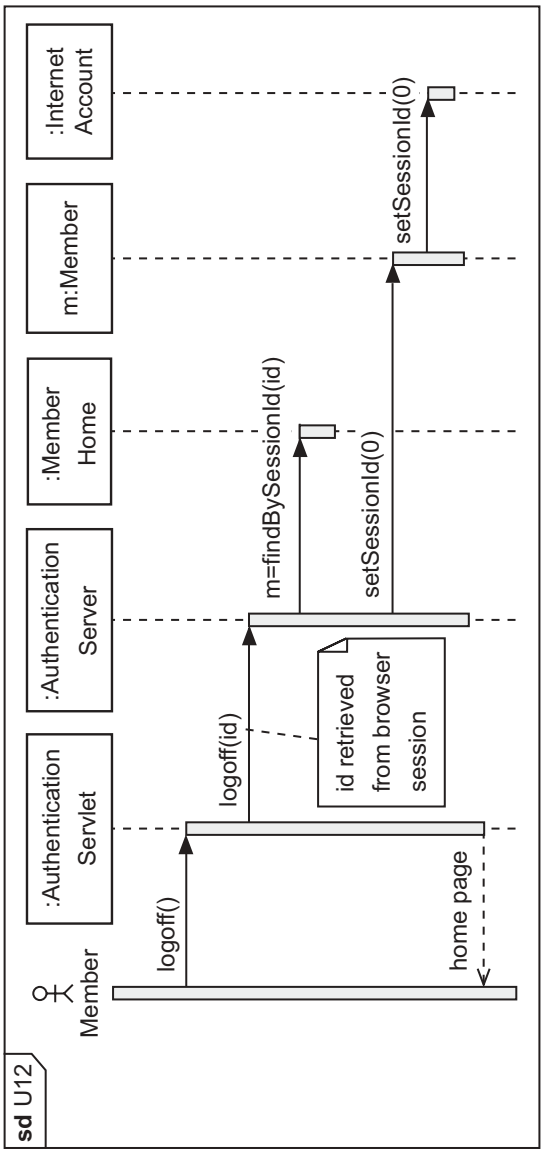


**Figure B.46: Sequence diagram for U12:Log Off**

# B.6 CLASS SPECIFICATION

This section documents the results of the specification phase of the iCoot development. Since there are many classes in iCoot, specifications are only included for one ServerLayer class and one BusinessLayer class, as examples. Each specification is given as comments in the source code.

## B.6.1 Server Class Specification

```
ReservationsServer Class Specification

Invariants: NONE
   // No invariants, because this is a stateless object

Methods:
   /*
    * Create a reservation for the member with session identifier i
    * and the car model with identifier c
    *
    * Preconditions:
    *    i != 0
    *    For mem = MemberHome.getInstance().findBySessionId(i),
    *       mem != null
    *       mem.isingoodStanding()
    *       mem.getAmountDue() == 0
    *    c != 0
    *    CarModelHome.getInstance().findByPrimaryKey(i) != null
    *
    * Postconditions:
    *    A new Reservation has been created for Member with session
    *    identifier i and CarModel with identifier c
    *
    * Exceptions:
    *    PServerException (checked) thrown if the server has a problem
    *    IllegalArgumentException (unchecked) thrown if parameters are
    *       invalid
    */
   public void createReservation(int i, int c)
      throws PServerException;

   /*
    * Read all reservations for the member with session identifier i
    *
    * Preconditions:
    *    i != 0
```

```
     *    MemberHome.getInstance().findBySessionId(i) != null
     *
     * Postconditions:
     *    result != null
     *    result contains all unconcluded reservations for Member
     *       with session identifier i
     *    result is a new array, exclusive to the client
     *
     * Exceptions:
     *    PServerException (checked) thrown if the server has a problem
     *    IllegalArgumentException (unchecked) thrown if parameters are
     *       invalid
     */
    public PReservation[] readReservations(int i)
       throws PServerException;

    /*
     * Cancel the reservation with identifier r for the member
     * with session identifier i
     *
     * Preconditions:
     *    i != 0
     *    For mem = MemberHome.getInstance().findBySessionId(i)
     *       mem != null
     *    r != 0
     *    For res = ReservationHome.getInstance().findByPrimaryKey(r)
     *       res != null
     *       res.getMember() == mem
     *
     * Postconditions:
     *    For res = ReservationHome.getInstance().findByPrimaryKey(r)
     *       res.isConcluded()
     *       res.getReason().equals("Canceled by customer");
     *
     * Exceptions:
     *    PServerException (checked) thrown if the server has a problem
     *    IllegalArgumentException (unchecked) thrown if parameters are
     *       invalid
     */
    public void cancelReservation(int i, int r)
       throws PServerException;
```

## B.6.2 Business Logic Class Specification

Member Class Specification

```
Invariants:
   /*
    * Values named in invariants, preconditions and postconditions
    * are attributes, with a getter and an optional setter. Each
    * invariant is an extra precondition for the corresponding setter
    * and an extra postcondition for the corresponding getter.
    */
   id is fixed after creation
   id != 0
   number != null
   number.size() != 0
   internetAccount != null
   address != null

Methods:
   /*
    * Fetch the receiver's id
    *
    * Preconditions: NONE
    * Postconditions:
    *    result == id
    * Exceptions: NONE
    */
   public int getId();

   /*
    * Fetch the receiver's number
    *
    * Preconditions: NONE
    * Postconditions:
    *    result == number
    * Exceptions: NONE
    */
   public String getNumber();

   /*
    * Set the receiver's number to n
    *
    * Preconditions: NONE
    * Postconditions:
    *    number == n
    * Exceptions: NONE
    */
   public void setNumber(String n);
```

```
/*
 * Fetch the receiver's internetAccount
 *
 * Preconditions: NONE
 * Postconditions:
 *    result == internetAccount
 * Exceptions: NONE
 */
public InternetAccount getInternetAccount();

/*
 * Set the receiver's internetAccount to ia
 *
 * Preconditions: NONE
 * Postconditions:
 *    internetAccount == ia
 * Exceptions: NONE
 */
public void setInternetAccount(InternetAccount ia);

/*
 * Fetch the receiver's address
 *
 * Preconditions: NONE
 * Postconditions:
 *    result == address
 * Exceptions: NONE
 */
public String getAddress();

/*
 * Set the receiver's address to a
 *
 * Preconditions: NONE
 * Postconditions:
 *    address == a
 * Exceptions: NONE
 */
public void setAddress(Address a);

/*
 * Fetch the session identifier of the receiver's internetAccount
 *
 * Preconditions: NONE
 * Postconditions:
```

```
 *    result == internetAccount.getSessionId()
 * Exceptions: NONE
 */
public int getSessionId();

/*
 * Set the session identifier of the receiver's
 *    internetAccount to i
 *
 * Preconditions: NONE
 * Postconditions:
 *    internetAddress.getSessionId() == i
 * Exceptions: NONE
 */
public void setSessionId(int i);
```

# B.7 OUTLINE TEST PLAN

## B.7.1 Introduction

The testing of iCoot will be continuous, with the involvement of developers, peers, customers, the build team and the testing team.

- Developers will test their artifacts as they produce them.
- Customers will be involved in the verification of high-level artifacts, acceptance testing and beta testing.
- Peers will review the artifacts produced by developers.
- The build team will be responsible for build testing after the first increment.
- The testing team will be responsible for coordinating the testing process, including the production and maintenance of this plan, the testing phase itself and system testing in particular.

This test plan gives an overview of the testing that will be carried out, followed by details of the testing tasks for each phase of development. It doesn't address the implementation of a prototype for iCoot. The development of a prototype will be under the supervision of project managers, conducted using rapid, informal methods.

## B.7.2 The Impact of Spirals and Increments

Within each spiral of development, testing will be carried out by the developers. Each artifact will be subject to peer review, with the proviso that formal peer reviews won't be necessary

for the first spiral. For every spiral after the second, formal peer reviews will concentrate on changes that have been made to the artifacts, to avoid duplicated effort. Similarly, customer reviews should concentrate on changes that have been made since the most recent review.

After each complete set of spirals, the testing team will take over to manage the testing phase before the release of the next increment.

After the first increment, regression testing will ensure that iCoot is at least as functional as it was after the previous increment.

### B.7.3 Testing of Non-Code Artifacts

Use cases and UML diagrams will be produced by the development team with input from customers. In the early spirals, members of the development team will review each other's work and changes will be made immediately.

During the last spiral before release of an increment, formal peer reviews will be held with colleagues who have relevant expertise but who are not directly involved with the project. These reviews will be used to certify the artifacts from use cases all the way through to class-based specifications.

Developers, peers and project managers will be responsible for ensuring that artifacts remain consistent over time.

### B.7.4 Code Reviews

After the final coding phase within each increment, formal code reviews will be held with peers who are not involved in the project itself. During these reviews, manual white-box testing and metric tools will be used to identify any refactoring that is needed.

### B.7.5 Test-Driven Development

During implementation of the design, programmers will perform continuous testing of their work with the help of JUnit. These programmer-developed tests will comprise unit tests at the class level and integration tests for all classes that the developer owns. Each developer will fix faults in their own code before making the code public.

At the end of each spiral, the developers will work together to perform integration testing and subsystem testing at a fairly informal level, again using JUnit, the aim being to eliminate as many faults as possible before the formal testing phase. The development team will fix any faults that they discover at this point, before the next spiral or the testing phase, as appropriate.

## B.7.6 Assertions

As detailed later in this test plan, programmers will be expected to add assertions to their code which will be enabled during development. During the testing phase, assertions will be initially enabled to make it easier to identify faults. Once all the tests are successful, they will be run again with assertions disabled, to check that none of the assertions have side effects. It will also allow performance with and without assertions to be compared. For release, assertions will be disabled but retained in the code so that they can be reactivated to help diagnose failures. So that the live system is not compromised by the disabling of assertions, further steps will be taken to protect iCoot.

In order to protect the servers from accidental or malicious attack when assertions are disabled, an application firewall will be implemented within the server layer. This firewall will largely enforce the server layer's preconditions in such a way that the checks can't be disabled. In order to reduce the amount of incorrect information that reaches the server layer, another application firewall will sit beneath the user interfaces (in the control layer) to reject invalid requests from the user. This client firewall will be based largely on the preconditions of the control layer, but again implemented in such a way that the checks can't be disabled. In addition, the Web server will be placed in a de-militarized zone, sandwiched between two conventional Internet firewalls to frustrate typical Internet attacks.

There will be two styles of client firewall. For HTML-based clients, the firewall will be implemented within the servlets and standard Web techniques for preventing invalid requests will be used (omitting invalid selections from Web pages and checking input data using JavaScript, for example). For GUI-based clients, the firewall will be implemented in the control layer and invalid requests will be prevented by using standard GUI techniques (such as disabling buttons that shouldn't be used and replacing text entry with drop-down lists and spin buttons).

## B.7.7 Testing Phase

Once the spirals for each increment have been completed, the code will be handed over to the testing team. Before formal testing begins, the testing team will re-run the unit and integration tests that have been produced by the development team, to verify that there are no known faults. The testing team will use the class and subsystem specifications to help them produce subsystem test cases, with accompanying test procedures. The system test cases will be based on the system use cases. Each test case will comprise a number of individual tests, each with test name, test description, test procedure and expected results.

The testing team will address the following requirements:

- Load testing, at average and maximum loads.
- Soak testing, to verify that there is no corruption or exhaustion of resources over time.

- Stress testing, to confirm that iCoot fails elegantly.
- Security testing.

Test automation will be used, wherever possible, to reduce the cost of testing.

In parallel, the testing team will organize acceptance testing, with the help of in-house volunteers and carefully selected customer personnel.

Installation testing will then be performed using a testbed that comprises a significant subset of the target platforms. Finally before release, beta testing will be performed at selected customer sites. All known faults will be fixed by the development team prior to release, wherever feasible.

Performance metrics (for example, average transaction time) and style metrics (for example, method size) will be gathered. Any unacceptable performance must be fixed prior to release. Style issues will be recorded for input to the next increment.

## B.7.8 Documentation Testing

After the first spiral, the documentation team will start to produce manuals and training materials. These will be subject to peer review. After several spirals, during the explicit testing phase that takes place before the release of each increment, the documentation will again be tested. This testing will comprise peer review, acceptance testing and beta testing.

## B.7.9 Build Testing

Programmers will use a source code management tool to ensure that all code is kept in a central location. They will be required to check out any code that they intend to work on, in order to avoid duplicated or overlapping effort. The project managers will be responsible for deciding who works on what. The build team will be responsible for managing the code repository. In order to cope with developer absences, the code management tool will be configured to allow checked-out artifacts to be checked back in, under controlled conditions.

After the first increment has been released, all further development will be subject to nightly build tests, run by the build team. This will involve building the entire system and running a significant subset of the system tests.

## B.7.10 Test Documentation and Logging

All test cases will be documented and kept in a test repository under the control of the testing team. As tests are carried out, test results will be recorded and added to the repository.

In order to encourage developers to test their own code before the testing phase, they won't have to record test failures in the test repository. Since the developers' test cases will be written in Java (using JUnit), they will be stored in the code repository; therefore, there will be no need to add such test cases to the test repository.

During the testing phase, test failures will be added to the repository. The testing team will collaborate with project managers to ensure that fixes are allocated to members of the development team and completed.

It will be the responsibility of project managers to ensure that adequate JUnit test cases are developed and run frequently. For reporting purposes, the development team must sign off the integration testing that takes place at the end of each spiral, by adding a corresponding entry to the test repository.

## B.7.11 Testing Activities by Phase

Set out below are the testing activities that will take place within each phase of development. The requirement for formal peer reviews will be relaxed for the first spiral. Peers will be selected on the basis of expertise.

- **Requirements phase** The business use cases, user interface sketches, the use case diagram, system use cases, use case priorities, supplementary requirements and activity diagrams (where used) will be reviewed by developers, peers and customers.
- **Analysis phase** The analysis class diagram, state machine diagrams (where used) and communication diagrams will be reviewed by developers, peers and customers.
- **System design phase** The deployment diagram, technology choices, layer diagram, layer interaction policy, concurrency policy and security policy will be reviewed by developers and peers.
- **Subsystem design phase** The design class diagrams; database schema; user interface design (for HTML access to servlets and JSPs, applets, applications, and interfaces for mobile devices); and sequence diagrams will be reviewed by developers and peers.
- **Specification phase** For each class, preconditions, postconditions and class invariants will be specified. These assertions will be reviewed by the developers and peers. Subsystem specifications may be used where appropriate. These will be tested in a similar way to class specifications.
- **Implementation phase** The testing during this phase consists of three parts:
  - Adding assertions (as per the class specification) to methods. Other types of assertion, such as checking loop termination and avoidance of impossible conditions, are optional, but recommended.
  - Creating JUnit test cases and JUnit test suites. There will be at least one test case (which is likely to involve some integration testing) per class and one test suite per package.
  - Performing code reviews, by developers and peers.
- **Testing phase** The testing phase will be the responsibility of the testing team. The development of test cases and test procedures by the testing team will be reviewed by peers. Testing will comprise:
  - The JUnit tests, to ensure that all known faults have been fixed.

  - Subsystem testing, based on any subsystem interfaces specified during the design and specification phases.
  - System testing, based on use cases (functional testing and load testing at average load and maximum load).
  - Stress testing, to confirm elegant failure (as defined during the design and specification phases).
  - Security testing (aggressive attempts to break into iCoot without authorization).
  - Acceptance testing, based largely on productivity metrics.
  - Metrics, based on system performance and coding style.
  - Documentation testing, with the help of end users and system administrators.
  - Installation testing, using a significant subset of target environments.
  - Beta testing, at selected customer sites.
- Maintenance phase The testing team will be responsible for managing the reporting and fixing of faults discovered after release, with the help of project managers and the development team. Between increments, fixes may be implemented, regression-tested and released at the discretion of the testing team and project managers. Regression testing will comprise:
  - The JUnit tests.
  - Subsystem testing.
  - System testing.
  - Installation testing.

  Feedback from customers about possible improvements to iCoot will be passed on to the project managers, with a view to incorporation in the next increment.

# B.8 GLOSSARY

| Term | Definition |
|---|---|
| Address (Business object, system object, analysis object, design object) | Where a Member lives. |
| AddressHome (Design object) | Home for creating and finding Address objects. |
| Assistant (Business actor, system actor) | An employee at a store who helps Customers to rent Car objects and reserve CarModels. |
| Auk (Business actor) | The pre-existing system that handles Customer details, Reservations, Rentals and the Catalog of available CarModels. |
| AukInterface (Analysis object) | Boundary for accessing Auk. |

| | |
|---|---|
| AuthenticationServer (Design object) | Controls the logging on and logging off of Members to iCoot. |
| AuthenticationServerHome (Design object) | Home for creating an AuthenticationServer. |
| AuthenticationServlet (Design object) | Makes the AuthenticationServer accessible in an HTML page in a Web browser. |
| BusinessLayer (Design layer) | Contains objects that convert the PersistenceLayer into clean object-oriented application objects. |
| Car (Business object, system object, analysis object, design object) | Instance of a CarModel for rent kept by a Store. |
| CarDetails (Analysis object, design object) | Extra details of a Car, such as number plate and VIN. |
| CarDetailsHome (Design object) | Home for creating and finding CarDetails. |
| CarHome (Design object) | Home for creating and finding Cars. |
| CarModel (Business object, system object, analysis object, design object) | A model in our Catalog, available for reservation. |
| CarModelDetails (Analysis object, design object) | Extra details about a CarModel, such as advert and poster. |
| CarModelDetailsHome (Design object) | Home for creating and finding CarModelDetails. |
| CarModelHome (Analysis object, design object) | Home for finding and creating CarModels. |
| Catalog (Business object) | A document describing CarModels available for rent. |
| CatalogQuery (Design object) | A Member's specification of CarModels that they're interested in when searching the iCoot on-line Catalog; includes categories, makes or engine sizes. |
| CatalogQueryHome (Design object) | Home for creating CatalogQuery objects. |
| CatalogServer (Design object) | Controls access to CarModels that can be browsed or reserved (Members only) over iCoot. |
| CatalogServerHome (Design object) | Home for creating a CatalogServer. |
| CatalogServlet (Design object) | Makes the CatalogServer accessible in an HTML page in a Web browser. |
| Category (Analysis object, design object) | Classification of a Car that helps Customers find what they're looking for, e.g. "Sports" or "Luxury". |

| | |
|---|---|
| CategoryHome (Design object) | Home for creating and finding Category objects. |
| Collectable (Design object) | A ReservationState indicating that a Customer has been informed about a matched Reservation but has not yet collected it. |
| com::nowhere::business (Design package) | Package containing the BusinessLayer classes. |
| com::nowhere::control (Design package) | Package containing the ControlLayer classes. |
| com::nowhere::micro (Design package) | Package containing the MicroLayer classes. |
| com::nowhere::persistence (Design package) | Package containing the PersistenceLayer classes. |
| com::nowhere::protocol (Design package) | Package containing the protocol classes, used by the ServletsLayer, RMILayer and ControlLayer for communication with the ServerLayer. |
| com::nowhere::rmi (Design package) | Package containing the RMILayer classes. |
| com::nowhere::server (Design package) | Package containing the ServerLayer classes. |
| com::nowhere::servlets (Design package) | Package containing the ServletsLayer classes. |
| com::nowhere::swing (Design package) | Package containing the SwingLayer classes. |
| Concluded (Design object) | A ReservationState indicating that a Reservation is finished because it was collected, canceled or timed out. |
| CootBusinessServer (Design node) | A process hosting iCoot services on a CootServer. |
| CootGUIClient (Design node) | A Customer's machine hosting a J2SE or J2ME GUI for accessing iCoot over RMI. |
| CootHTMLClient (Design node) | A Customer's machine hosting a Web browser to access iCoot. |
| cootschema.ddl (Deployment artifact) | Script used to generate database tables for Coot. |
| CootServer (Design node) | A machine hosting a WebServer and CootBusinessServer. |
| CreditCard (Business object, system object, analysis object, design object) | Used for confirming the credit-worthiness of Members; must not have expired for a Member to be in good standing. |
| CreditCardCompany (Business actor) | Company that issues CreditCards and confirms validity. |

| | |
|---|---|
| CreditCardHome (Design object) | Home for creating and finding CreditCards. |
| Customer (Business actor, business object, system actor, system object, analysis object, design object) | A person who pays money in return for one of our standard services. |
| DBMS | A process hosting a relational database management system. |
| DBServer | A machine hosting a DBMS. |
| DebtDepartment (Business object) | The department that deals with unpaid fees. |
| Displayable (Design object) | A ReservationState indicating that a Reservation that was Collectable has timed out or been canceled; means that a Car is in the Reserved area that must be moved back to the display area. |
| EJB | An Enterprise JavaBean; an object within a standard Java framework that can handle transactions, network access and database access, behind an Internet firewall; these come in two varieties, session beans (for remote access to business services) and entity beans (for automatic mapping of data to and from a database). |
| HTMLLayer (Design layer) | The client-side code for accessing the ServletsLayer; provided by the standard HTML Web browser. |
| HTTPCGILayer (Design layer) | The standard network layer that sits between the HTMLLayer and the ServletsLayer. |
| icoot.ear (Deployment artifact) | Java enterprise archive containing the servlets, JSPs and EJBs used by CootHTMLClients and, eventually, CootGUIClients. |
| iCoot (Deployment artifact) | Folder containing static HTML for the iCoot site. |
| IllegalArgumentException | Standard Java Exception that indicates an attempt was made to send a message with invalid parameters. |
| InternetAccount (Design object) | Details required for a Member to log on to iCoot plus a record of their logged-on status. |
| InternetAccountHome (Design object) | Home for creating and finding InternetAccounts. |

| | |
|---|---|
| J2EE | Enterprise Edition of the Java 2 platform. |
| J2ME | Micro Edition of the Java 2 platform. |
| J2SE | Standard Edition of the Java 2 platform. |
| JDBC | A standard Java library that provides access to all relational databases in a uniform way. |
| JDBCLayer (Design layer) | Layer that accesses a relational database from Java (within the EJB framework). |
| JRMP | The communication protocol used by RMI. |
| JSP | Java Server Page; a dynamic web page containing Java code, to be executed by the server, alongside static HTML. |
| Keys (Business object) | For operating a Car; Customers have copies when they're renting; Store keeps copies for available Cars and reserve copies for all Cars; Store has serial number for reproduction by Make if all copies are lost or broken. |
| LegalDepartment (Business Actor) | The department that deals with accidents in which a rented Car has been involved. |
| License (Business object) | A document that must be presented in order to rent a Car or as proof of identity when a NonMember makes a Reservation. |
| LogonController (Analysis object) | iCoot controller that controls logging on and off by Members. |
| Make (Analysis object, design object) | Each Car has one or more Makes that manufactures it. |
| MakeHome (Design object) | Home for creating and finding Makes. |
| MemberHome (Analysis object, design object) | Home for finding and creating Members. |
| MembershipCard (Business object) | A laminated document issued by a store to a Member as proof of membership. |
| MembershipServer (Design object) | Controls access to a member's details, such as Address and CreditCard, over iCoot. |
| MembershipServerHome (Design object) | Home for creating a MembershipServer. |
| MembershipServlet (Design object) | Makes the MembershipServer accessible within an HTML page in a Web browser. |
| MemberUI (Analysis object) | iCoot boundary used by Members to access the system. |

| | |
|---|---|
| MicroLayer | Objects using the Java 2 Micro Edition to access the RMILayer from a pervasive device, such as a mobile phone or set-top box; reserved for future versions of iCoot. |
| NeedingRenewal (Design object) | A ReservationState indicating that the Reservation has not been matched for a week and must be renewed if it is not to expire. |
| NonMember (Business actor, business object, system actor, analysis object, design object) | A Customer whose identity and creditworthiness have not been checked and who, therefore, must provide a deposit to make a Reservation or surrender a copy of their License to rent a Car. |
| NonMemberUI (Analysis object) | iCoot boundary used by NonMembers to access the system. |
| Notifiable (Design object) | A ReservationState indicating that a Reservation has been matched to an available Car but the Member has not yet been notified. |
| PAddress (Design object) | Protocol version of Address. |
| PAddressHome (Design object) | Home for creating InternetAccounts. |
| PCar (Design object) | Protocol version of Car. |
| PCarHome (Design object) | Home for creating PCar objects. |
| PCarModel (Design object) | Protocol version of CarModel. |
| PCarModelDetails (Design object) | Protocol version of CarModelDetails. |
| PCarModelDetailsHome (Design object) | Home for creating PCarModelDetails. |
| PCarModelHome (Design object) | Home for creating PCarModels. |
| PCatalogQuery (Design object) | Protocol version of CatalogQuery. |
| PCatalogQueryHome (Design object) | Home for creating PCatalogQuery objects. |
| PCategory (Design object) | Protocol version of Category. |
| PCategoryHome (Design object) | Home for creating a PCategory. |
| PCreditCard (Design object) | Protocol version of CreditCard. |
| PCreditCardHome (Design object) | Home for creating PCreditCards. |
| PMake (Design object) | Protocol version of Make. |
| PMakeHome (Design object) | Home for creating PMakes. |
| PMember (Design object) | Protocol version of Member. |
| PMemberHome (Design object) | Home for creating PMembers. |
| PRental (Design object) | Protocol version of Rental. |
| PRentalHome (Design object) | Home for creating PRentals. |
| PReservation (Design object) | Protocol version of Reservation. |
| PReservationHome (Design object) | Home for creating PReservations. |

| | |
|---|---|
| PServerException (Design object) | An indication that one of the objects in the ServerLayer has been unable to complete a request because of, for example, a database problem. |
| PServerExceptionHome (Design object) | Home for creating PServerExceptions. |
| Rental (Business object, system object, analysis object, design object) | A contract between Nowhere Cars and a Customer to keep one or more Cars for an agreed period; subject to late fees if Car is not returned on time. |
| RentalHome (Analysis object, design object) | Home for finding and creating Rentals. |
| RentalServerHome (Design object) | Home for creating a RentalServer. |
| RentalsServer (Design object) | Controls access to a Member's Rentals over iCoot. |
| RentalsServlet (Design object) | Makes the RentalServer accessible within an HTML page in a Web browser. |
| Reservation (Business object, system object, analysis object, design object) | The reserving of a CarModel by a Customer. |
| ReservationHome (Analysis object) | Home for finding and creating Reservations. |
| ReservationServerHome (Design object) | Home for creating a ReservationServer. |
| ReservationsServer (Design object) | Controls access to Reservations for Members over iCoot. |
| ReservationsServlet (Design object) | Makes the ReservationServer accessible within an HTML page in a Web browser. |
| ReservationsSlip (Business object) | A slip detailing the membership number, car model, timestamp and number for a Reservation. |
| ReservationState (Analysis object, design object) | The state of a reservation, e.g. Waiting or Concluded. |
| ReservationStateHome (Design object) | Home for creating a ReservationState. |
| RMI | Standard Java mechanism for sending messages to an object over a network. |
| RMILayer (Design layer) | Converts the ServerLayer into simple objects that can be accessed by the SwingLayer or MicroLayer; reserved for future versions of iCoot. |
| ServerLayer | Contains objects to access iCoot over a network. |
| ServletsLayer (Design layer) | The server-side objects that provide access to iCoot from an HTML Web browser. |

| | |
|---|---|
| Store (Business actor, design object) | A Nowhere Cars site from which Cars can be rented, CarModels reserved and Catalogs browsed or obtained. |
| SwingLayer (Design layer) | Objects using the standard Java library for accessing the RMILayer from a Java GUI; reserved for future versions of iCoot. |
| Vendor (Analysis object, design object) | Company that supplies one or more Cars. |
| VendorHome (Design object) | Home for creating and finding Vendor objects. |
| VIN | Vehicle Identification Number; unique number issued by the licensing authority and appearing on a plate riveted to the Car's body. |
| Waiting (Design object) | A ReservationState indicating that the Reservation has been made over the Internet but has not yet been satisfied, canceled or expired. |
| WebBrowser (Design node) | A process providing HTML access to a CootHTMLClient. |
| WebServer (Design node) | A process providing server-side access to a CootBusinessServer from a Web browser. |