Prof. Dr. A. Podelski, Dr. B. Westphal
C. Schätzle Wintersemester 2016/17

Software Design, Modeling, and Analysis in UML

`http://swt.informatik.uni-freiburg.de/teaching/WS2016-17/sdmauml`

Exercise Sheet 6.B

Early submission: Tuesday, 2017-01-25, 12:00 Regular submission: Tuesday, 2017-01-25, 12:00

Exercise 1 (10/20 Points)

There is a Rhapsody model of a web shop software available for download in the ILIAS which is yet lacking the behaviour of class Session (cf. appendix).

(i) Create a class diagram of the existing classes and the associations between them which gives a good overview of the overall system structure. Include most relevant attributes and methods at your discretion. (2)

(ii) Provide a state machine for class Session realising the behaviour specified in the appendix.

Give an indication of the suitability of your design by providing (a) test case(s) for at least the following use-cases (briefly describe the expected behaviour according to the specification):

a) The frontend is requested to add an item to a non-existing session.
b) A new session is created via a connection to the frontend.
c) The frontend is requested to add a non-existing item to an existing session.
d) Proceeding to checkout is requested via the frontend for a session without any items in the shopping cart.
e) The frontend is requested to add an existing item to an existing session.
f) Proceed to checkout and return to shopping via the frontend.
g) Have a session with at least two items added to the shopping cart via the frontend.
h) A credit card number is submitted to a session in the checkout state via the frontend.
i) An invalid security number is submitted to a session awaiting it via the frontend.
j) A valid security number is submitted instead.

(7)

Hint: Make appropriate use of features of hierarchical state machines. Provide an appropriate description of the resulting web-shop model, which in particular describes your design decisions (e.g. where to use which hierarchical state machine feature) on the Session's behaviour (see note below).

(iii) What is the customer experience if an invalid credit card number, i.e. one for which there is no valid security number, is submitted? (1)

Note: Exercise Sheet 7.B will be a peer review of the submissions to this exercise sheet (in anonymised form). Consider the tasks of Exercise Sheet 7.B for the review criteria and prepare your submissions accordingly.

Exercise 2 (5 Bonus)

Customers may close their web browser at any time, even in the middle of shopping or checkout. The existing design would keep the corresponding Session object as long as the software is running, which consumes unnecessary resources. The customer therefore requires a timeout mechanism to be implemented: if a Session instance does not receive any events for a given amount of time, it should request the backend to release all of its reservations for items, deregister at the frontend, and destroy itself[1] if the frontend confirms deregistration.

Propose an extension of your design for such a timeout mechanism. For simplicity, we do not model the time-keeping but only make the Session's state machine sensitive to a new signal 'Timeout' which is treated as an environment signal (i.e. will only be observed when explicitly sending it via Rhapsody's event injection dialog).

To ensure that the bookkeeping on reserved and available items at the backend is kept consistent, the Session object should not be destroyed when it just has a pending request on an item. For simplicity, assume that Timeout events will be re-sent by the timeout mechanism until the Session object is able to process one, thus Timeout events can safely be discarded if they arrive at an inconvenient time.

Provide recorded sequence diagrams demonstrating your timeout handling for at least the two cases that a Session object has and has not a pending request on an item.

Hint: Make appropriate use of features of hierarchical state machines. Provide an appropriate description of the resulting web-shop model, which in particular describes your design decisions (e.g. where to use which hierarchical state machine feature) on the Session's behaviour.

Exercise 3 (5 Bonus)

The author of the web shop Rhapsody model claims that the apparently over-complex procedure of deregistering at the frontend and waiting for a reply (instead of the Session object just destroying itself) is necessary to avoid reachability of the error configuration (in case of Rhapsody: abnormal termination of the program).

Is this claim true? If yes, sketch (or provide) a counter-example and explain what goes wrong, If no, argue why.

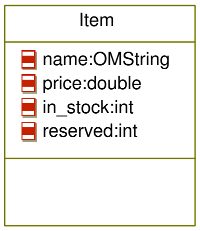
[1] It seems as if Rhapsody does not treat final states in the way we discussed in the lecture. You may use the explicit *termination pseudo-state* (an uppercase T in a green circle) for self destruction.

Appendix

The Web-Shop Model

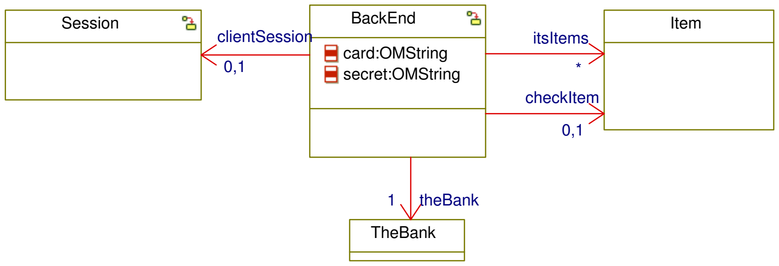
The following sections give an overview of the existing web shop model, the available classes including their attributes and associations, and basic functionality. Details like available operations can be looked up in the Rhapsody model.
Note that none of the existing classes (except for TheBank (see below)) should be changed; subject of the exercise is to provide a state machine for class Session.

Item



Objects of class Item are used to keep track of the items in stock. There is one Item object for each kind of items, which has a name, a price, a number of items of this item in stock, and a number of items reserved by currently shopping customers. When an item is bought, the number in stock and the number of reservations is decremented by 1. Class Item does not have a statechart.

BackEnd



Each web shop has exactly one instance of BackEnd (for simplicity, there is no class for web shops; the initialisation code creates exactly one BackEnd instance, i.e. exactly one web shop).
A BackEnd has a list of available items (itsItems). The global function `initItems` creates two kinds of items, 'ABX427' and 'ZCY539'.[2]
BackEnd instances have two temporary associations checkItem and clientSession, which are used when processing requests by Session instances. Each BackEnd has exactly one instance of TheBank, which is consulted during the payment procedure.
BackEnd offers three services.

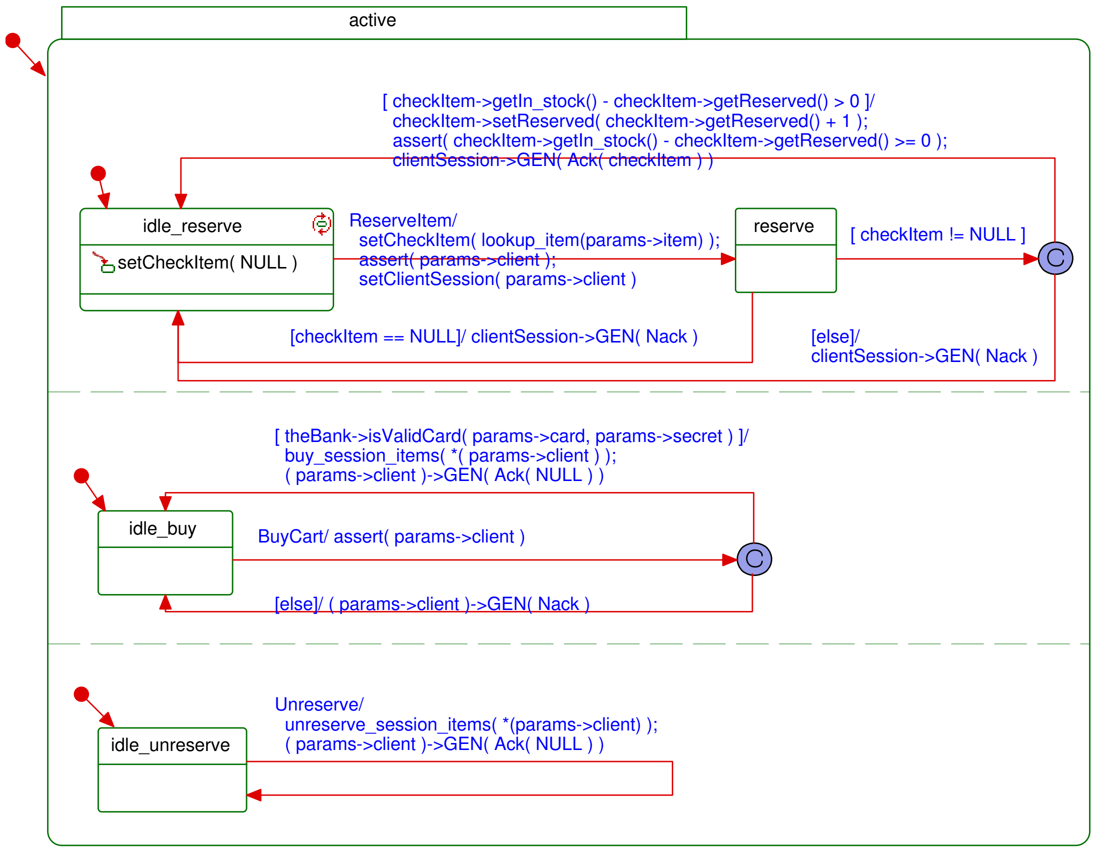
- On event ReserveItem (which carries two parameters: a string giving the name of an item to be reserved, and a pointer to the Session object posting the request), the given item name is looked up and, if the item exists and there are more items in stock than already reserved, an Ack (carrying a pointer to the corresponding Item instance) is sent to the Session which posted the request. In addition, the reservation counter of the corresponding Item instance is incremented. Otherwise, a Nack event is sent back.
- On event BuyCart (which carries three parameters: a string giving the credit card number, a string giving the security code, and a pointer to the Session object posting the request), it is checked with the TheBank instance, whether the combination of card number and security code is valid. If yes, for each item of the Session object, the number of items in stock and the number of reserved items are

[2] You may edit this function to sell other items, as many as you like.

decremented. The list of items of the Session object is cleared and an Ack event is sent back. (For simplicity, we don't keep track of the credit card's balance.) Otherwise, a Nack event is sent back.

- On event Unreserve (which carries one parameter: a pointer to the Session object posting the request), the reservation counter of all items of the session is decremented, the list of items of the session is cleared, and an Ack event is sent back.

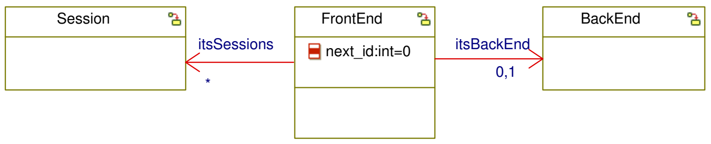
  This service should be used when a Session instance is about to be destroyed while having items in the shopping cart, e.g., due to a timeout.


active
[checkItem->getIn_stock() - checkItem->getReserved() > 0]/
checkItem->setReserved(checkItem->getReserved() + 1);
assert(checkItem->getIn_stock() - checkItem->getReserved() >= 0);
clientSession->GEN(Ack(checkItem))
idle_reserve
setCheckItem(NULL)
ReserveItem/
setCheckItem(lookup_item(params->item));
assert(params->client);
setClientSession(params->client)
reserve
[checkItem != NULL]
C
[checkItem == NULL]/ clientSession->GEN(Nack)
[else]/
clientSession->GEN(Nack)
[theBank->isValidCard(params->card, params->secret)]/
buy_session_items(*(params->client));
(params->client)->GEN(Ack(NULL))
idle_buy
BuyCart/ assert(params->client)
C
[else]/ (params->client)->GEN(Nack)
Unreserve/
unreserve_session_items(*(params->client));
(params->client)->GEN(Ack(NULL))
idle_unreserve

Bank

Instances of TheBank don't have a statemachine. They basically offer the operation `isValidCard` which takes a string giving a credit card number and a string giving a security code, and return `true` if and only if the given combination is valid. [3]

FrontEnd



Class FrontEnd models the interface to the customers, i.e. it may create web pages to be delivered to the customers' web-browsers showing the list of items in the shop, asking for payment information and payment confirmation etc. Actions on the customer's side are forwarded to the corresponding Session object. Instances of FrontEnd may have a link to a BackEnd instance, which is passed on to newly created Session instances, and a link to a set of active Session instances.

[3] In the given model, there is only one credit card with number `12345678` and security code `910`. You may edit this function to offer other or more valid credit cards.

FrontEnd offers the following services:

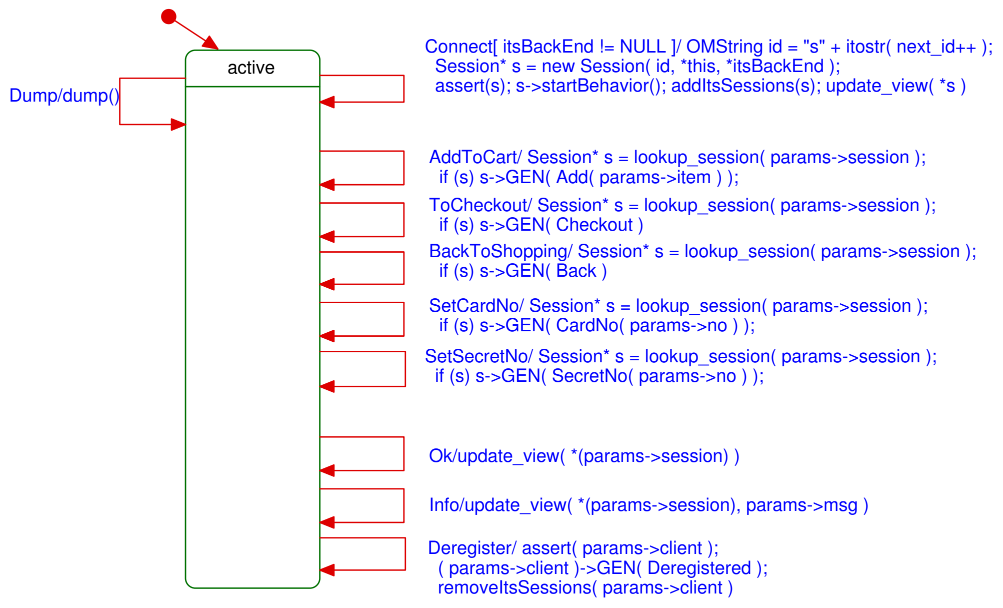
- On event Connect, a new Session instance is created with a new unique string identifier (if a BackEnd is available). The customer's browser is supposed to provide the identifier with following requests, thereby FrontEnd may serve multiple customers at the same time.
- On event AddToCart (which carries two parameters: a session identifier string and an item identifier string (we assume, that the web page delivered to the customer lists items on sale together with these identifiers)) sends an Add event to the session denoted by the identifier (if it exists).
- Session objects are supposed to reply to forwarded events by either sending event Ok or Info. Both cause method `update_view` to be called, which, in the given model, for simplicity prints out a dump of the sessions state or an information message. These actions correspond to updating the view on the customer's side.
- On event ToCheckout (carrying one parameter: a session identifier), a Checkout event is sent to the session denoted by the identifier (if it exists).

  This events corresponds to the customer selecting 'checkout' on the web shop to initiate the payment process.
- On BackToShopping, a Back event is sent to the session (if it exists).

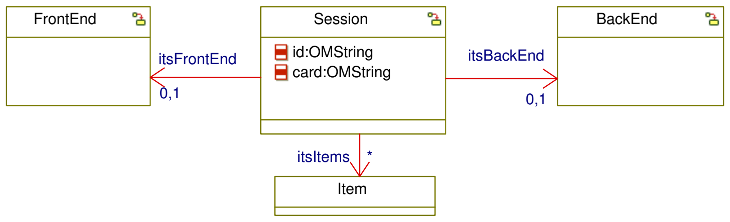
  This event corresponds to the customer selecting 'continue shopping' at the payment page instead of giving credit card information.
- On SetCardNo, a CardNo event is sent to the session, passing on the given credit card number.

  This event corresponds to the customer submitting a credit card number on the payment page.
- On SetSecretNo, a SecretNo event is sent to the session, passing on the given security code.

  This event corresponds to the customer submitting a security code after having initiated the payment with SetCardNo.
- Event Deregister (carrying one parameter: a pointer to a session object) is supposed to be sent by Session instances before they destroy itself (either after completing the payment, or on a timeout). The FrontEnd removes the session from the list of existing sessions and replies with a Deregistered event.
- Event Dump can be sent for debugging purposes. It prints out the status of all existing sessions and the BackEnd instance (if available).


active
Dump/dump()
Connect[itsBackEnd != NULL]/ OMString id = "s" + itostr(next_id++);
Session* s = new Session(id, *this, *itsBackEnd);
assert(s); s->startBehavior(); addItsSessions(s); update_view(*s)
AddToCart/ Session* s = lookup_session(params->session);
if (s) s->GEN(Add(params->item));
ToCheckout/ Session* s = lookup_session(params->session);
if (s) s->GEN(Checkout)
BackToShopping/ Session* s = lookup_session(params->session);
if (s) s->GEN(Back)
SetCardNo/ Session* s = lookup_session(params->session);
if (s) s->GEN(CardNo(params->no));
SetSecretNo/ Session* s = lookup_session(params->session);
if (s) s->GEN(SecretNo(params->no));
Ok/update_view(*(params->session))
Info/update_view(*(params->session), params->msg)
Deregister/ assert(params->client);
(params->client)->GEN(Deregistered);
removeItsSessions(params->client)

Session



Instances of class Session may have a link to a FrontEnd by which it has been created (to reply to requests and to deregister), and a link to a BackEnd to request reservation of items, and to buy the items in the shopping cart.
The shopping cart is modelled by association itsItems, which is supposed to hold a set of pointers to Item instances.
Session already has a method `dump` which prints out the current content of the shopping cart (and overall cost so far). You may edit this method to also print out the currently assumed screen as modelled by the states of the state machines (cf. implementation of `dump` in the Rhapsody model).

Session Behaviour: Requirements Specification

A session is supposed to basically support three screens at the customers' side:

- The shopping phase (which is also the initial phase), where the customers select items and add them to their shopping cart.
- The counter, where the customers are supposed to provide credit card information.
- The credit card security check, where customers are supposed to submit the credit card's security code.

For the shopping screen, the Session should process Add events which carry an item identifier.

- For simplicity, we want to allow items to be in the shopping cart at most once (and we do not support removing items from the cart). That is, if the item is already in the shopping cart, the Session object should reply to its FrontEnd by action

  ```
  itsFrontEnd− >GEN( Info( this, "Item in cart." ) )
  ```

  Whether an item is already in the shopping cart can be checked by action

  ```
  lookup_item()
  ```

  which takes an item identifier and returns NULL if and only if the corresponding item is not yet in the shopping cart.

- Otherwise, a request for reserving this item should be sent to its BackEnd.
- If the BackEnd replies with event Nack (identifier does not denote an item, or there are not items available for reservation), the Session object should reply to its FrontEnd by action

  ```
  itsFrontEnd− >GEN( Info( this, "No such item." ) )
  ```

- If the BackEnd replies with Ack (carrying a pointer to the corresponding Item object), the Session object should reply to its FrontEnd by action

  ```
  itsFrontEnd− >GEN( Ok( this ) )
  ```

  and add the obtained Item pointer to the shopping cart.

Furthermore, the Session should process Checkout events. If the shopping cart is not empty, the shopping screen should be left for the counter (and reply Ok to the FrontEnd). Otherwise, it should reply by

```
itsFrontEnd− >GEN( Info( this, "Cart is empty." ) )
```

and remain on the shopping screen.

For the payment screen, the Session should be able to process Back events, just reply with Ok, and go back to the shopping screen.
Furthermore, it should process CardNo, and (if the given credit card number is not empty (as checked by guard `!params− >no.IsEmpty()`)), assign the obtained credit card number to its `card` attribute, reply by Ok, and change to the security check screen.

For the security check screen, the Session should be able to process SecretNo events. A BuyCart event should be sent to the BackEnd, including the (previously stored) card number and the newly obtained security code. If the reply from the BackEnd is a Nack (card not valid), the security check screen should again be assumed allowing the user to correct the problem by sending the security code anew.
If the reply from the BackEnd is an Ack, the Session should just reply with an Ok to the FrontEnd. Afterwards, a designated state should be assumed which does not process any further events (or initiates deregistering and self-destruction when doing the bonus task).

No other behaviour than the ones described above should be present (unless there is a good (and in your submission well-described) reason).