SWE 621: Software Modeling and Architectural Design

Lecture Notes on Software Design

Lecture 11 - Detailed Software Design

Hassan Gomaa
Dept of Computer Science
George Mason University
Fairfax, VA

SWE 621: Software Modeling and Architectural Design

Lecture 11 - Detailed Software Design

Hassan Gomaa

Reference: H. Gomaa, Chapters 14 - *Software Modeling and Design*, Cambridge University Press, February 2011

Reference: H. Gomaa, Chapter 16 - *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley Object Technology Series, July, 2000

Copyright © 2011 Hassan Gomaa
All rights reserved. No part of this document may be reproduced in any form or by any means, without the prior written permission of the author.
This electronic course material may not be distributed by e-mail or posted on any other World Wide Web site without the prior written permission of the author.

Steps in Using COMET/UML

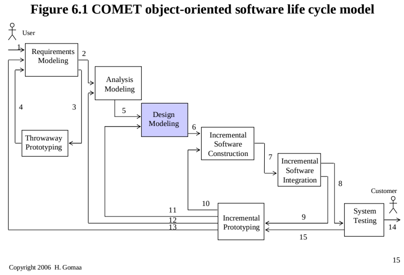
1 Develop Software Requirements Model

2 Develop Software Analysis Model

3 **Develop Software Design Model**

- Design Overall Software Architecture (Chapter 12, 13)
- Design Distributed Component-based Subsystems (Chapter 12-13,15)
- Structure Subsystems into Concurrent Tasks (Chapter 18)
- Design Information Hiding Classes (Chapter 14)
- **Develop Detailed Software Design**

Figure 6.1 COMET object-oriented software life cycle model



Detailed Software Design

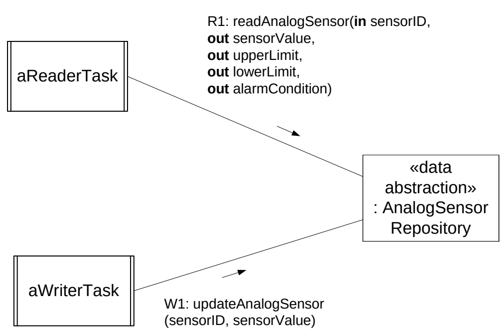
- Design details of task synchronization
  - Passive objects accessed by more than one task
- Design connector classes
  - Address details of inter-task communication
- Define each task's internal event sequencing logic
  - Pseudocode description

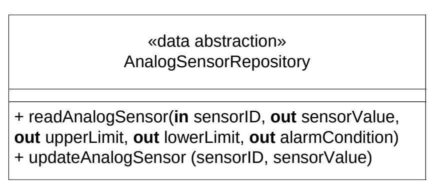
Synchronization of Tasks Interacting via Passive Objects

Task interaction via shared data
- Needs synchronization

Task interaction via passive data abstraction object
- Hides structure of data repository
- Hides synchronization from tasks
  - Mutual exclusion
  - Multiple readers / multiple writers

Example of concurrent access to data abstraction object





Information Hiding Objects Synchronization of Access

- Each information hiding object
  - Designed for application
- Mutually exclusive access to data repository
  - Use binary semaphore
- Access by multiple readers / writers
  - Allows access to data repository
    - By many readers concurrently
    - Only one writer

Interaction Between Concurrent Tasks

- Mutual exclusion
  - Two or more tasks need to access shared data
  - Access must be mutually exclusive
- Binary semaphore
  - Boolean variable that is only accessed by means of two atomic (indivisible) operations
  - **acquire (semaphore)**
    - if the resource is available, then get the resource
    - if resource is unavailable, wait for resource to become available
  - **release (semaphore)**
    - signals that resource is now available
    - if another task is waiting for the resource, it will now acquire the resource

Example of Mutual Exclusion

readAnalogSensor (**in** sensorID, **out** sensorValue, **out** upperLimit, **out** lowerLimit, **out** alarmCondition)

-- Critical region for read operation.

acquire (sensorDataStoreSemaphore)

  sensorValue := sensorDataStore (sensorID, value)

  upperLimit := sensorDataStore (sensorID, upLim)

  lowerLimit := sensorDataStore (sensorID, loLim)

  alarmCondition := sensorDataStore (sensorID, alarm)

release (sensorDataStoreSemaphore)

end readAnalogSensor

Example of Mutual Exclusion

updateAnalogSensor (**in** sensorID, **in** sensorValue)

-- Critical region for write operation.

acquire (sensorDataStoreSemaphore)

  sensorDataStore (sensorID, value) := sensorValue

  if sensorValue >= sensorDataStore (sensorID, upLim)

    then sensorDataStore (sensorID, alarm) := high

  elseif sensorValue <= sensorDataStore (sensorID, loLim)

    then sensorDataStore (sensorID, alarm) := low

    else sensorDataStore (sensorID, alarm) := normal

  endif;

release (sensorDataStoreSemaphore)

end updateAnalogSensor

Example of
Multiple Readers / Multiple Writers

```
readAnalogSensor (in sensorID, out sensorValue, out upperLimit, out lowerLimit, out alarmCondition)
-- Read operation called by reader tasks. Several readers are allowed
-- to access the data store providing there is no writer accessing it.
acquire (readerSemaphore)
    Increment numberOfReaders
    if numberOfReaders = 1 then acquire (sensorDataStoreSemaphore)
release (readerSemaphore)
    sensorValue := sensorDataStore (sensorID, value)
    upperLimit := sensorDataStore (sensorID, upLim)
    lowerLimit := sensorDataStore (sensorID, loLim)
    alarmCondition := sensorDataStore (sensorID, alarm)
acquire (readerSemaphore)
    Decrement numberOfReaders
    if numberOfReaders = 0 then release (sensorDataStoreSemaphore)
release (readerSemaphore)
end readAnalogSensor
```


Example of
Multiple Readers / Multiple Writers

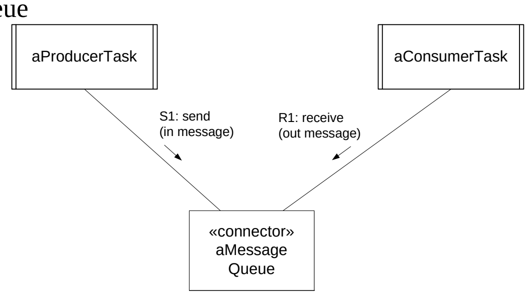
```
updateAnalogSensor (in sensorID, in sensorValue)
-- Critical region for write operation.
acquire (sensorDataStoreSemaphore)
    sensorDataStore (sensorID, value) := sensorValue
    if sensorValue >= sensorDataStore (sensorID, upLim)
            then sensorDataStore (sensorID, alarm) := high
    elseif sensorValue <= sensorDataStore (sensorID, loLim)
            then sensorDataStore (sensorID, alarm) := low
            else sensorDataStore (sensorID, alarm) := normal
    endif;
release (sensorDataStoreSemaphore)
end updateAnalogSensor
```

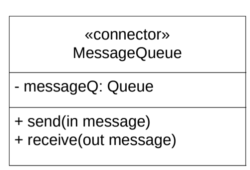
Connector Classes

– Classes designed to provide inter-task communication and synchronization
  – Asynchronous (loosely coupled) message communication
  – Synchronous (tightly coupled) message communication without reply
  – Synchronous (tightly coupled) message communication with reply
– Message buffering monitor classes
  – Synchronized (mutually exclusive) operations

Example of message queue connector object

• Asynchronous (loosely coupled) message communication
  - Use message queue monitor class
  - Encapsulates message queue





Synchronization within Connector Object

- Synchronization between tasks (Java threads)
  - When task enters synchronized operation, it acquires semaphore
  - Synchronization methods
    - Wait
      - Task is suspended, releases semaphore
    - Signal (Notify in Java)
      - Wake up a suspended task
  - Condition wait
    - Check condition for waiting, e.g.,
      - **while** messageCount = 0 **do wait**

Message Queue Connector Class

Monitor MessageQueue

-- Encapsulates a message queue that holds at most maxCount messages.

-- Monitor operations are executed mutually exclusively.

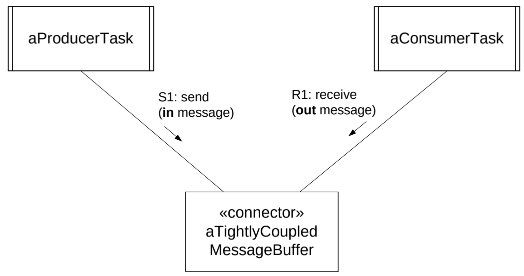
public send (**in** message)
    while messageCount = maxCount **do wait;**
    place message in buffer;
    Increment messageCount;
    if messageCount = 1 **then signal;**
end send;

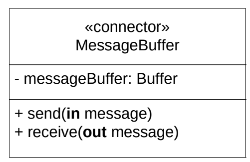
public receive (**out** message)
    while messageCount = 0 **do wait;**
    remove message from buffer;
    Decrement messageCount;
    if messageCount = maxCount-1 **then signal;**
end receive;

end MessageQueue;

Example of message buffer connector

- Synchronous (tightly coupled) message communication without reply
  - Encapsulates a message buffer
  - Holds at most one message





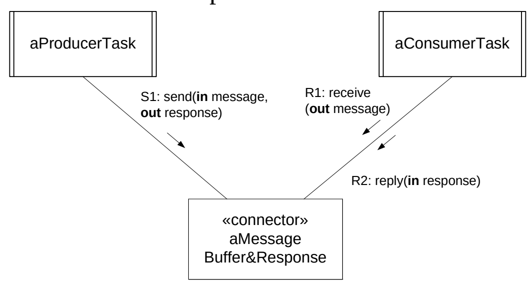
Message Buffer Connector Class

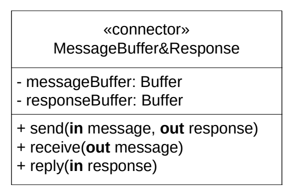
```
monitor MessageBuffer
    -- Encapsulates a message buffer that holds at most one message.
    -- Monitor operations are executed mutually exclusively.
public send (in message)
    place message in buffer
    messageBufferFull := true
    signal;
    while messageBufferFull = true do wait;
end send;

public receive (out message)
    while messageBufferFull = false do wait;
    remove message from buffer
    messageBufferFull := false
    signal;
end receive;
end MessageBuffer;
```

Example of message buffer and response connector

- Synchronous (tightly coupled) message communication with reply
  - Encapsulates a message buffer - Holds one message
  - Encapsulates a response buffer - Holds one response





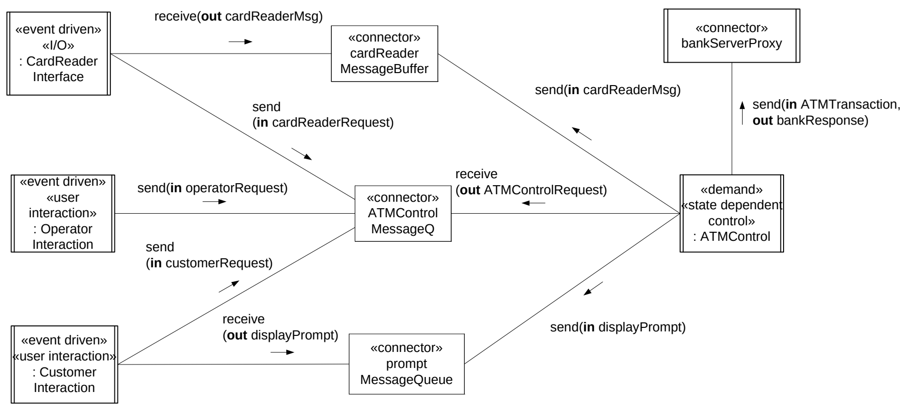
Message Buffer & Response Connector Class

```
monitor MessageBuffer&Response
 -- Encapsulates a message buffer, which can hold at most one message
 -- and a response buffer, which can hold at most one response.
 -- Monitor operations are executed mutually exclusively.
public send (in message, out response)
     place message in buffer;
     messageBufferFull := true;
     signal;
     while responseBufferFull = false do wait;
     remove response from response buffer;
     responseBufferFull := false;
     return (response);
end send;

public receive (out message)
     while messageBufferFull = false do wait;
     remove message from buffer;
     messageBufferFull := false;
end receive;

public reply (in response)
     Place response in response buffer
     responseBufferFull := true
     signal;
end reply;
end MessageBuffer&Response;
```

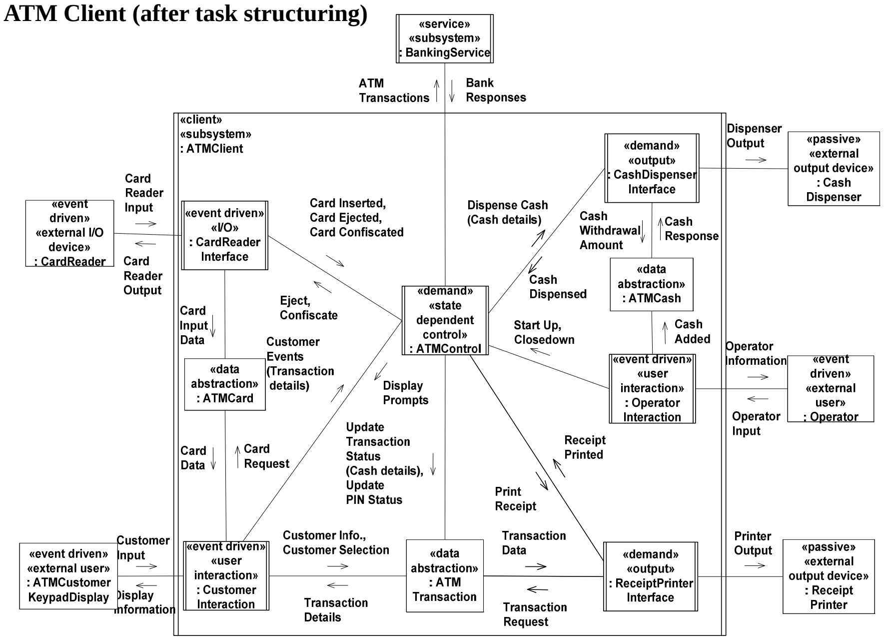
Example of cooperating tasks using connectors



Task Behavior Specifications (TBS)

- Developed during Detailed Software Design
- Define each task's internal event sequencing logic
  - Describe informally in Pseudocode
  - Describe task response to each message or event input

Figure 18.13 Task architecture – initial concurrent communication diagram for ATM Client (after task structuring)



Event Sequencing Logic for Card Reader Interface Task

```
-- Initialize Card Reader;
cardReaderDI.initialize();
loop
-- Wait for external interrupt from card
reader
wait (cardReaderEvent);
-- Read card data held on card's magnetic
strip
cardReaderDI.read (cardInput);
if card recognized
then -- Write card data to ATM Card
object;
  ATMCard.write (cardData);
  -- send card Inserted message to ATM
Controller
  ATMControlMessageQ.send (cardInserted);
  -- Wait for message from ATM Controller
  cardReaderMessageBuffer.receive
(message);
  if message = eject
  then -- Eject card
      cardReaderDI.eject ();
      -- Send card Ejected message to ATM
Controller
      ATMControlMessageQ.send
(cardEjected);
  elseif message = confiscate
      then -- confiscate card
        cardReaderDI.confiscate ();
        -- Send card Confiscated message
to ATM Controller;
        ATMControlMessageQ.send
```

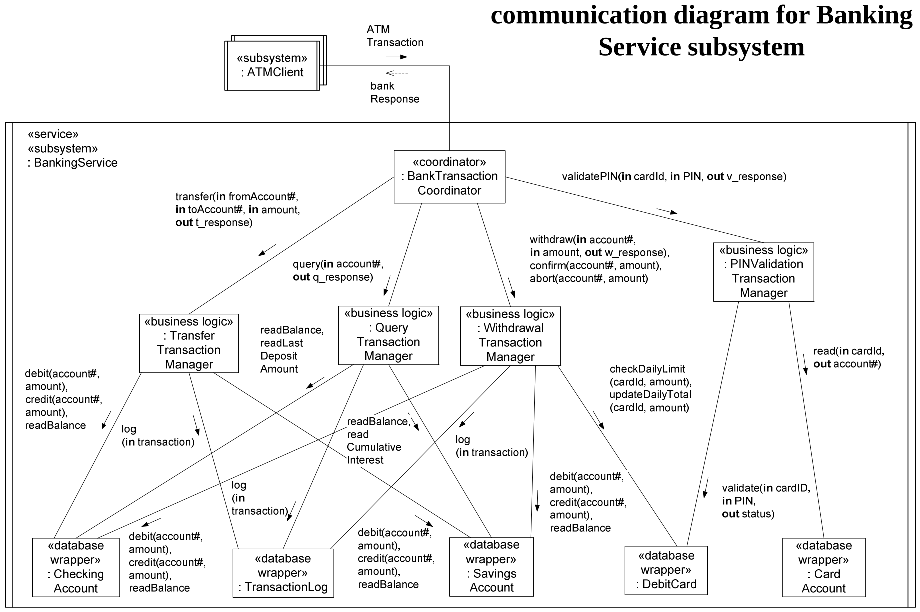
Event Sequencing Logic for ATM Controller Task

```
loop
 -- Messages from all senders are received on Message
Queue
 ATMControlMessageQ.receive (message);
 -- Extract the event name and any message parameters
 -- Given the incoming event, lookup state transition
table;
 -- change state if required; return action to be
performed;
 newEvent = message.event
 outstandingEvent = true;
 while outstandingEvent do
  ATMControl.processEvent (in newEvent, out action);
  outstandingEvent = false;
  -- Execute action(s) as given on ATM Control
statechart
  case action of
   Get PIN: -- Prompt for PIN;
     promptMessageQueue.send (displayPINPrompt);
   Validate PIN: --Validate customer entered PIN at
bank server;
     bankServerProxy.send (in validatePIN, out
validationResponse);
     newEvent = validationResponse; outstandingEvent =
true;
   Display Menu: -- Display selection menu to customer;
     promptMessageQueue.send (displayMenu);
     ATMTransaction.updatePINStatus (valid);
   Invalid PIN Action: -- Display Invalid PIN prompt;
     promptMessageQueue.send (displayInvalidPINPrompt);
        ATMTransaction.updatePINStatus (invalid);
   Request Withdrawal: -- Send withdraw request to bank
```


Event Sequencing Logic for ATM Controller Task

```
   Request Query: -- Send query request to bank
server;
     promptMessageQueue.send (displayWait);
     bankServerProxy.send (in queryRequest, out
queryResponse);
     newEvent = queryResponse; outstandingEvent =
true;
   Request Transfer: -- Send transfer request to bank
server;
     promptMessageQueue.send (displayWait);
     bankServerProxy.send (transferRequest, out
transferResponse);
     newEvent = transferResponse; outstandingEvent =
true;
   Dispense: -- Dispense cash and update transaction
status;
     ATMTransaction.updateTransactionStatus
(withdrawalOK);
     cashDispenserInterface.dispenseCash
       (in cashAmount, out dispenseStatus);
     newEvent = cashDispensed; outstandingEvent =
true;
   Print: -- Print receipt and send confirmation to
bank server;
     promptMessageQueue.send (displayCashDispensed);
     bankServerProxy.send (in confirmRequest);
     receiptPrinterInterface.printReceipt
        (in receiptInfo, out printStatus);
     newEvent = receiptPrinted; outstandingEvent =
true;
   Eject: -- Eject ATM card;
     cardReaderMessageBuffer.send (eject);
   Confiscate: -- Confiscate ATM card;
```

Figure 21.35 Revised concurrent communication diagram for Banking Service subsystem



Event Sequencing Logic for Banking Service Task

```
loop
receive (Client, Message) from Banking Service
Message Queue;
Extract message name and message parameters from
message;
case Message of
 Validate PIN:
   -- Check that ATM Card is valid and that PIN
entered by
   -- customer matches PIN maintained by Server;
   PINValidationTransactionManager.ValidatePIN
      (in CardId, in PIN, out validationResponse);
   -- If successful, validation Response is valid
and return
   -- Account Numbers accessible by this debit card;
   -- otherwise validation Response is invalid,
   -- third Invalid, or stolen;
   reply (Client, validationResponse);
 Withdrawal:
   -- Check that daily limit has not been exceeded
and that
   -- customer has enough funds in account to
satisfy request.
   -- If all checks are successful, then debit
account.
   WithdrawalTransactionManager.withdraw
```

Event Sequencing Logic for Bank Server Task

```
 Query:
   -- Read account balance.
   queryTransactionManager.query
      (in accountNumber, out queryresponse);
   -- Query Response = Current Balance and either
Last Deposit
   -- Amount (checking account) or Interest (savings
acount);
   reply (client, queryResponse);
 Transfer:
   -- Check that customer has enough funds in From
Account to
   -- satisfy request. If approved, then debit From
Account
   -- and credit To Account;
   transferTransactionManager.transfer (in
fromAccount#,
      in toAccount#, in amount, out
transferResponse);
   -- If approved, then transfer Response is
   -- {successful, Current Balance of From Account};
   -- otherwise Transfer Response is {unsuccessful};
   reply (client, transferResponse);
 Confirm:
   -- Confirm withdrawal transaction was completed
successfully
   withdrawalTransactionManager.confirm (in
```


Steps in Using COMET/UML

1 Develop Software Requirements Model

2 Develop Software Analysis Model

3 **Develop Software Design Model**

- Design Overall Software Architecture (Chapter 12, 13)
- Design Distributed Component-based Subsystems (Chapter 12-13,15)
- Structure Subsystems into Concurrent Tasks (Chapter 18)
- Design Information Hiding Classes (Chapter 14)
- **Develop Detailed Software Design**

Figure 6.1 COMET object-oriented software life cycle model