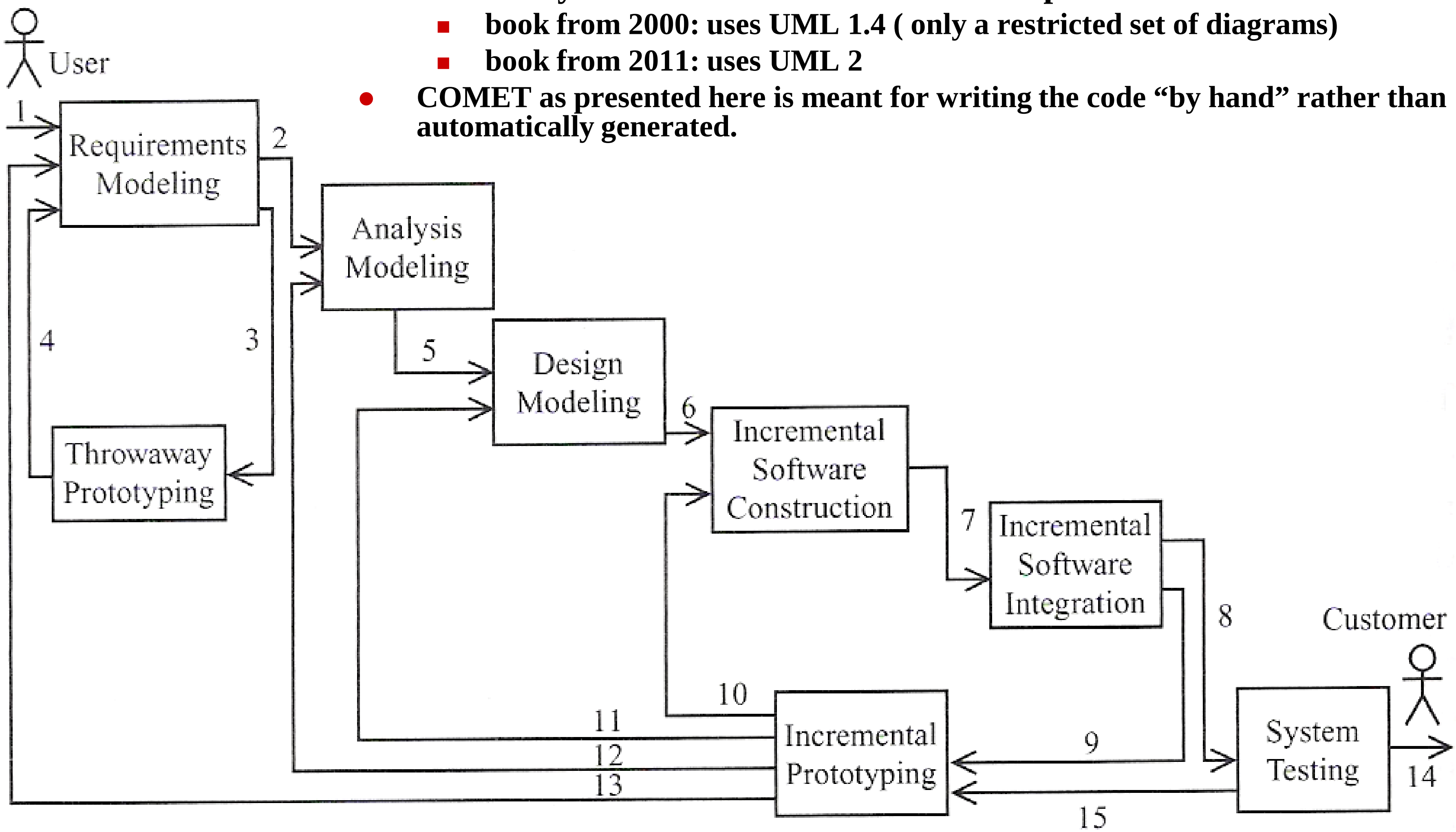
COMET- Design methodology for real-time & concurrent applications

COMET software life-cycle

- **COMET was developed by Hassan Gomaa for real-time distributed concurrent systems**
- **2 books by Gomaa listed in course outline presents COMET:**
  - **book from 2000: uses UML 1.4 (only a restricted set of diagrams)**
  - **book from 2011: uses UML 2**
- **COMET as presented here is meant for writing the code "by hand" rather than automatically generated.**



Step 1: Requirement Modeling

- Develop the use case model
  - develop use case diagram(s)
    - actors, use-cases and their relationships
  - textual description of use-cases
    - follow template
      - use case name and summary
      - actors
      - dependency of other use cases (e.g., what is included)
      - preconditions
      - narrative description of the "happy path"
      - description of different alternatives
      - postconditions

Step 2: Analysis Model (problem domain)

- **Develop static model (i.e., structure)**
  - **identify physical objects/classes in the problem (application) domain**
  - **develop system context model – interaction with external classes**
  - **entity classes – data intensive classes that store data (represent physical objects)**
  - **build a class dictionary (classes and attributes)**
- **Develop system structure: group classes into subsystems**
- **Develop dynamic model (i.e., behaviour). For each use case:**
  - **identify participant objects (classes)**
  - **develop interaction diagrams**
    - **describe "happy path" and all alternatives**
    - **identify information passed between objects**
  - **develop a statechart for each state-dependent object in a collaboration**
    - **events and actions in statechart are consistent with messages received and sent by the respective object in the collaboration**

COMET Step 3: Design Model (solution domain)

1. **Synthesize initial software architecture from the analysis model – this is a transition from analysis to design**
   - synthesize statecharts for each state-dependent object from the partial statecharts built in the analysis phase
     - each partial SC: the behaviour of the object in a collaboration
   - consolidate all the collaboration diagram in a collaboration model for the system – verify consistency
   - synthesize design static model
     - refinement of analysis static model – new objects needed for the solution may be added.
2. **Design overall software architecture**
   - structure application into subsystems
   - define interfaces between subsystems
   - develop collaboration diagrams for each subsystem and a high-level collaboration diagram for the whole system.

Step 3: Design Model (continued)

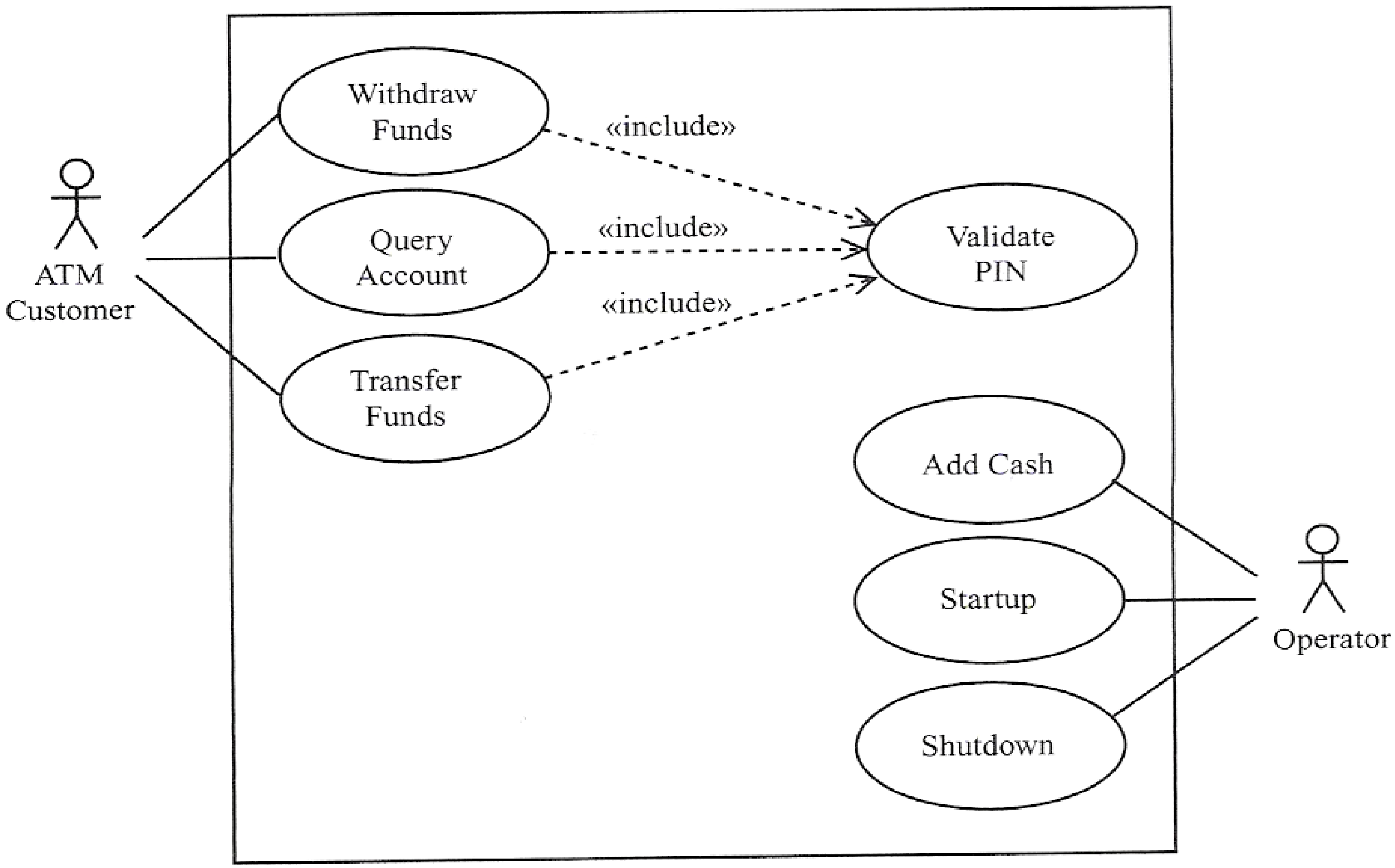
3. Design the distributed software architecture
   - identify distributed components
   - design their interfaces
4. Design concurrent task architecture for each subsystem
   - decide on the concurrent tasks and their interfaces
   - for each subsystem: develop concurrent collaboration diagram between its concurrent tasks
5. Analyze the performance of the design
   - predictive performance analysis by using performance models
6. Detailed software design for each subsystem
   - design internals of composite tasks – contain passive objects
   - design details of task communication/synchronization
7. Analyze the performance of the real-time design for each subsystem

Case Study: Banking System

- **Problem statement:**
  - **a bank has several automated teller machines (ATM) connected to a central bank server**
  - **each ATM has: card reader, cash dispenser, keyboard/ display, receipt printer**
  - **a customer may withdraw cash from a checking or saving account, query the balance, or transfer funds**
  - **a transaction is initiated when the customer inserts an ATM card into the card reader**
  - **customer authentication:**
    - **based on PIN – allows only for three attempts only**
    - **the info from the card is verified against the data maintained by the system – cards reported lost or stolen are confiscated**
  - **customer transaction may proceed after successful authentication**
  - **at the end, the customer record, account record and card record are updated at the bank server**
  - **an ATM operator may start up and close down the ATM to replenish the cash dispenser and for routine maintenance**
  - **simplifying assumptions: opening and closing accounts, adding/removing customers are not part of this problem**

Case Study step 1: Use Case Model

- **Actors in real-time embedded systems can be not only human users, but also external devices, external systems, timers, etc.**
  - **in this case the actors are human users**



Validate PIN Use Case: textual description

- **Use case name:** validate PIN
- **Summary:** system validates customer PIN
- **Actor:** ATM Customer
- **Preconditions:** ATM is idle, displaying a welcome message
- **Description:**
  1. Customer inserts the ATM card into the card reader
  2. System reads the card
  3. System prompts customer for PIN
  4. System check expiration date and whether the card is lost or stolen
  5. If card is valid, system check PIN validity against value stored in the system
  6. If PIN matches, system check if the card may access the account
  7. System displays customer account and prompts customer foe transaction type

Validate PIN Use Case (cont)

- **Alternatives:**
  1. If card not recognized, the system ejects the card
  2. If card expired, the system confiscates the card
  3. if card has been reported lost or stolen, the system confiscates it
  4. if the customer-entered PIN does not match the one stored by the system, the system re-prompts for PIN
  5. if the customer enters an incorrect PIN three times, the card is confiscated
  6. If the customer enters "Cancel" the transaction is cancelled and the card is ejected.
- **Postcondition:** Customer PIN has been validated.

Step 2: Analysis Model
Static model (i.e., structure)

- **Develop static model**
  - identify physical objects/classes in the problem domain
  - develop system context model
  - entity classes – data intensive classes that store data
  - build a class dictionary (classes and attributes)

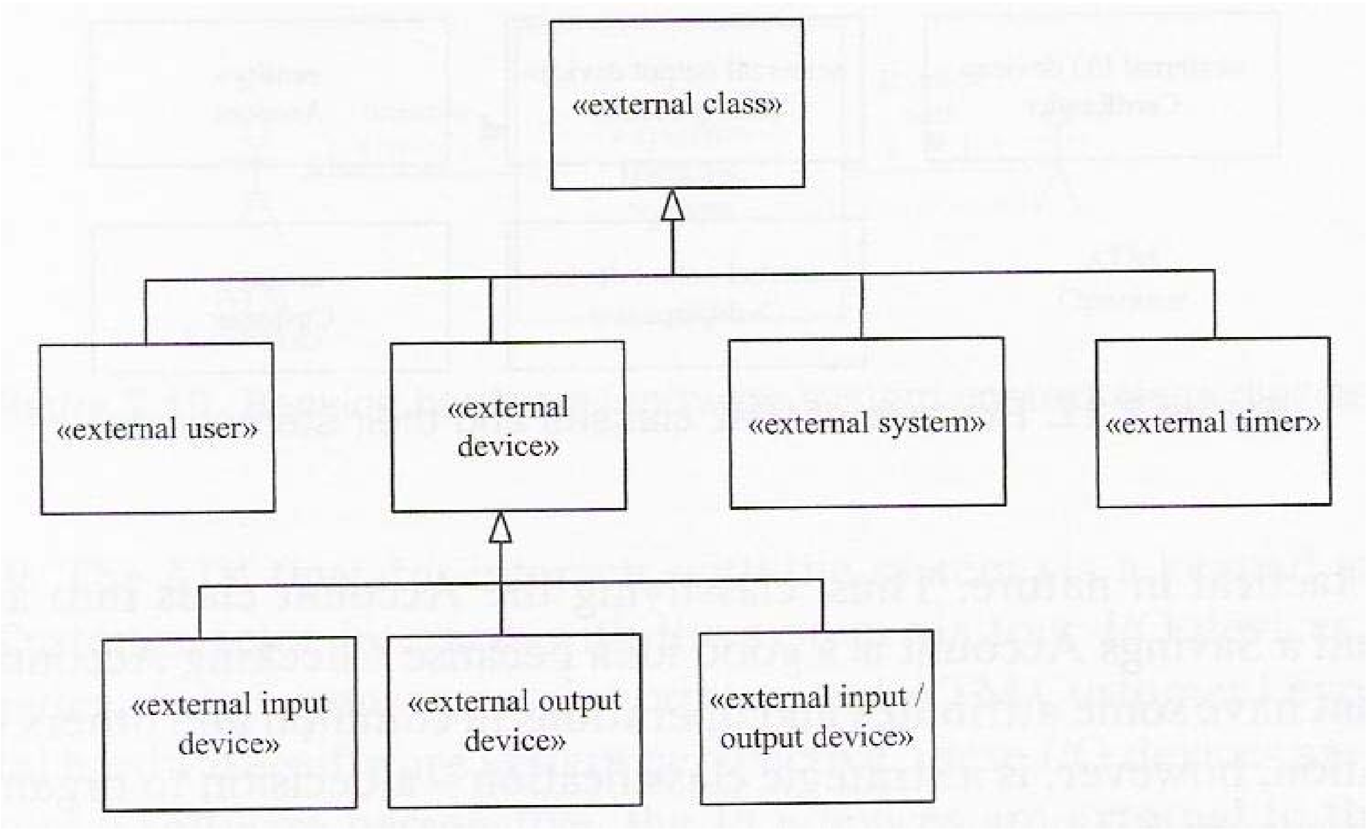
- **Develop system structure: start grouping classes into subsystems**
  (subsystems are finalized in the Design Model)

Domain Model

- **Domain model**: identifies **key concepts from a certain area/domain**
- Example: electro cardiogram (ECG) system model:
  - problem domain concepts: heart rate, arrhythmias, waveforms, scaling in time, scaling in amplitude
  - design concepts: data buffers, tasks and threads, semaphores
- Every application domain has its own vocabulary and concepts
  - some domains are closer to the problem space
  - other domains are closer to the solution space (implementation)
- A complete application may involve multiple domains which may be layered:
  - top-level domains belong to the application, while lower level domains represent the underlying platforms (OS, communications, hardware);
  - domains are normally stable, which foster reuse;
  - help to make the system development robust to change (of platform, of requirements, etc.) by limiting the impact of change.
- **COMET static analysis**: concerned with the **problem domain model**.
- Domain modeling advice: avoid introducing design concepts in the problem domain model.

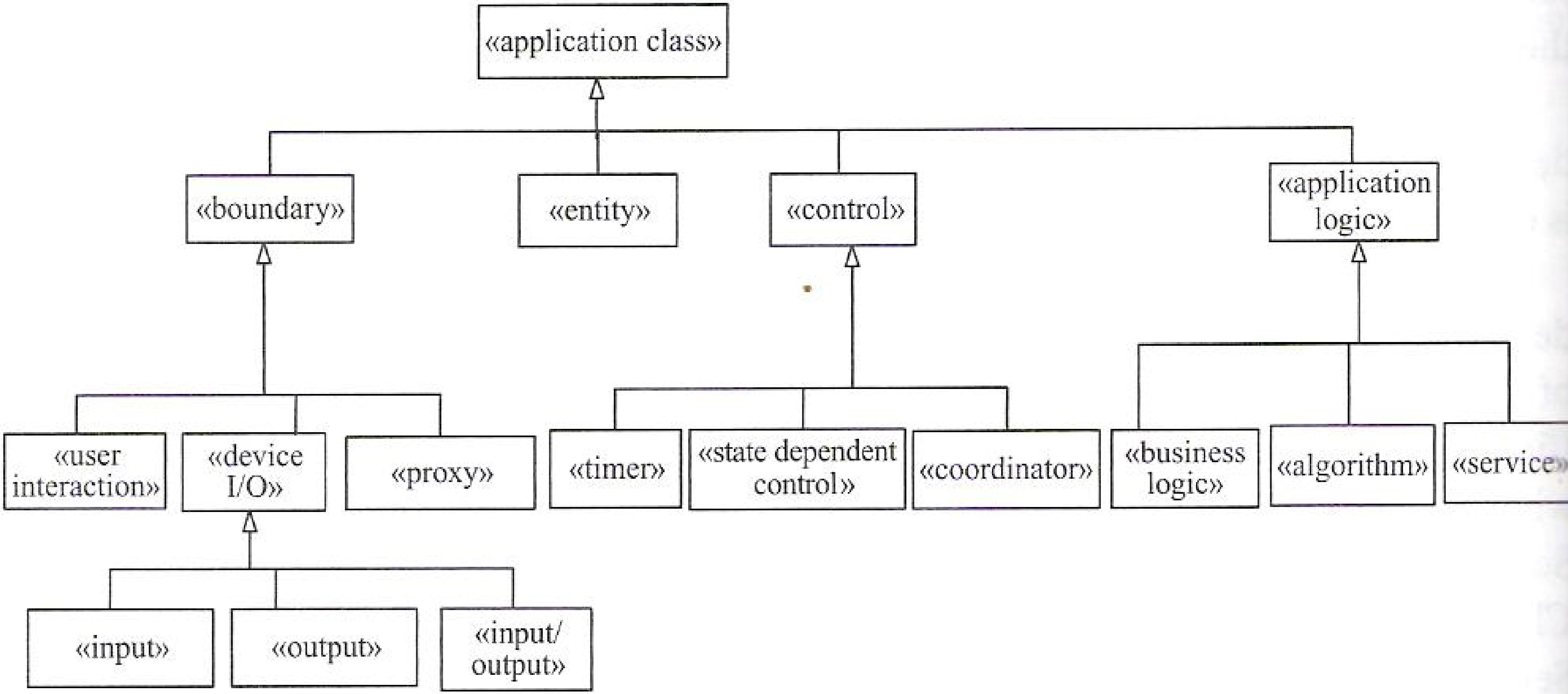
Categorizing external classes

- Identify separately and categorize by using UML stereotypes the following kind of classes:
  - application classes which are part of the system to be built (discussed in next slide)
  - external classes which are part of the environment:
    - «external user»: user interacting with the system and exchanging information via standard I/O devices (keyboard, screen, mouse - handled by the operating system)
    - «external device»: application-specific hardware devices (e.g., sensors, actuators)
    - «external system»: other systems interacting with our system
    - «external timer»: clock to keep track of time or timer to initiate timer events



Categorizing application classes (1)

- **application classes are part of the system under construction**
- **they are categorized according to the role played in the application (described on the next slide)**
- **the stereotype hierarchy shown below applies to classes as well as their instances.**



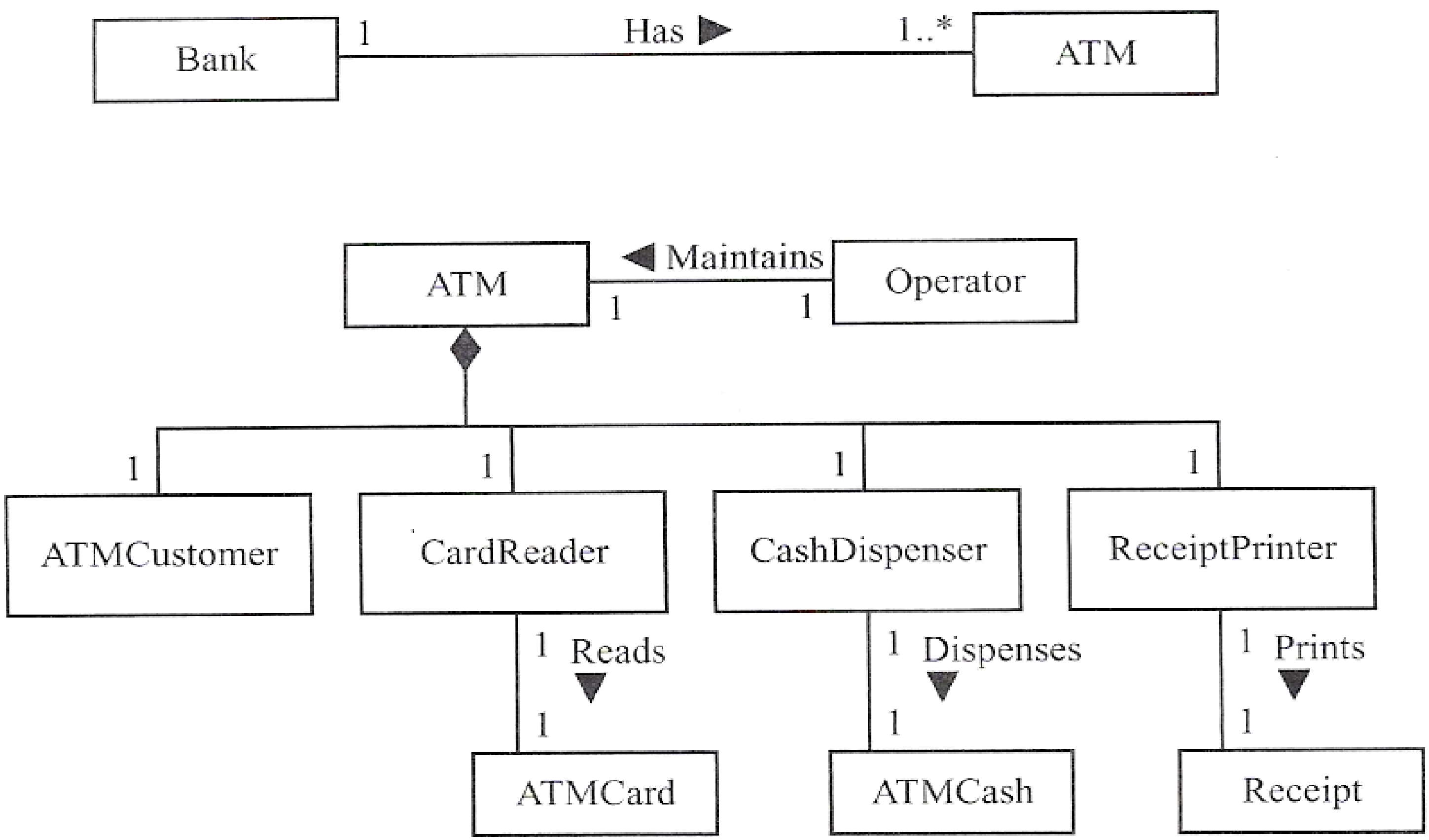
Categorizing application classes (2)

- **The class structuring categories are as follows:**
  1. **«entity»: encapsulates information and provides access to it (may be persistent)**
  2. **«boundary»: interfaces and communicates with the external environment**
     - **«user interaction»: interfaces with a human user via standard I/O devices**
     - **«device I/O»: interfaces with a hardware I/O device**
       - **may be «input», «output», or «input/output»**
     - **«proxy»: interfaces with an external system or subsystem.**
  3. **«control»: provides the overall coordination for a collection of objects**
     - **may be «state dependent control», «coordinator», or «timer»**
  4. **«application logic»: contains the details of the application logic; needed to separate the application logic from the data it manipulate**
     - **may be: «business logic» in business application, «algorithm» in scientific applications, or «service» in service-based systems.**

COMET static Analysis

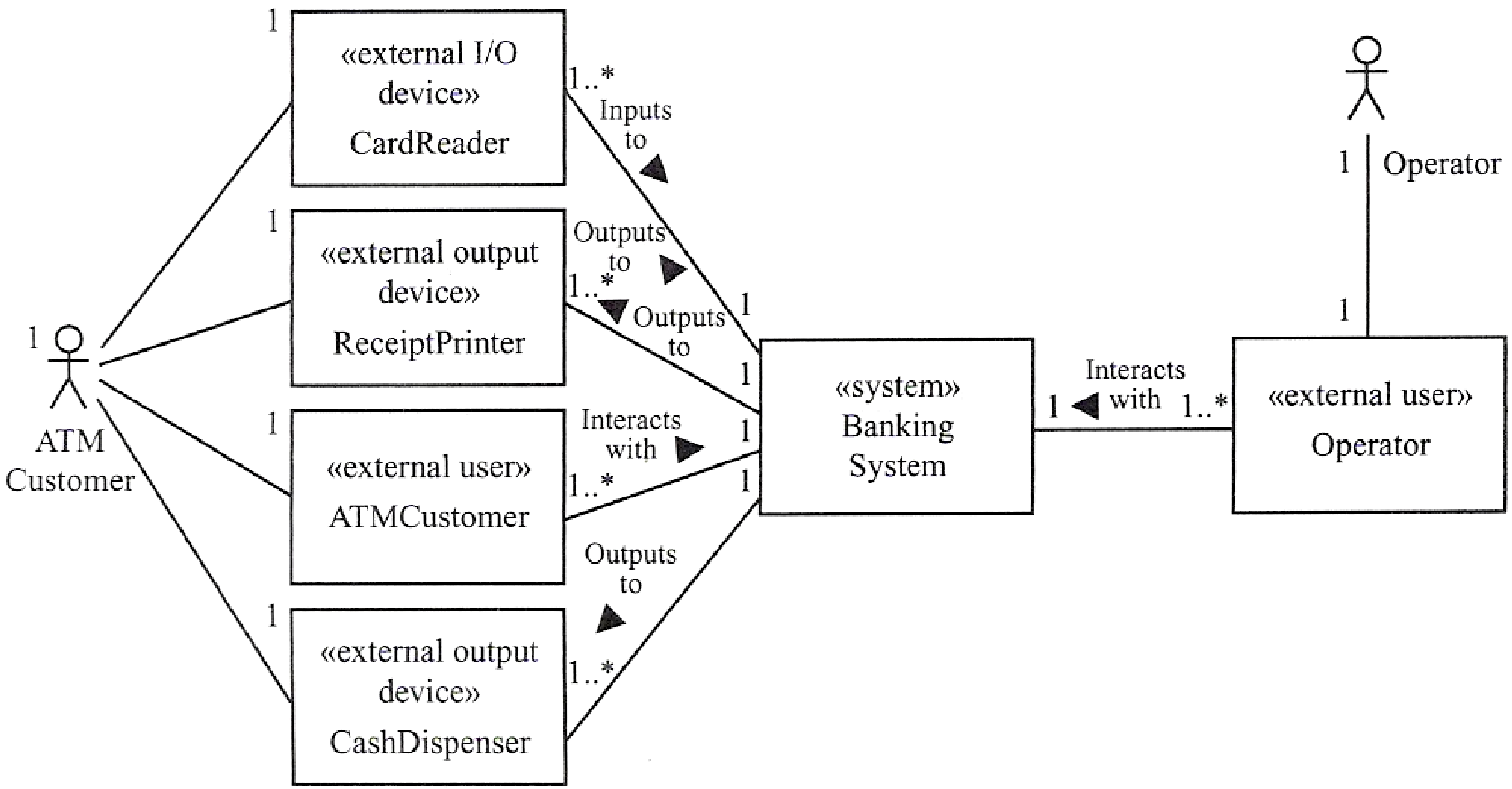
Problem domain: physical classes

- conceptual static model of the problem domain
- identify physical classes and their relationships
  - many physical objects end up being represented in the software as “entity classes”
  - some are “external users” – which represent both the user and its interface



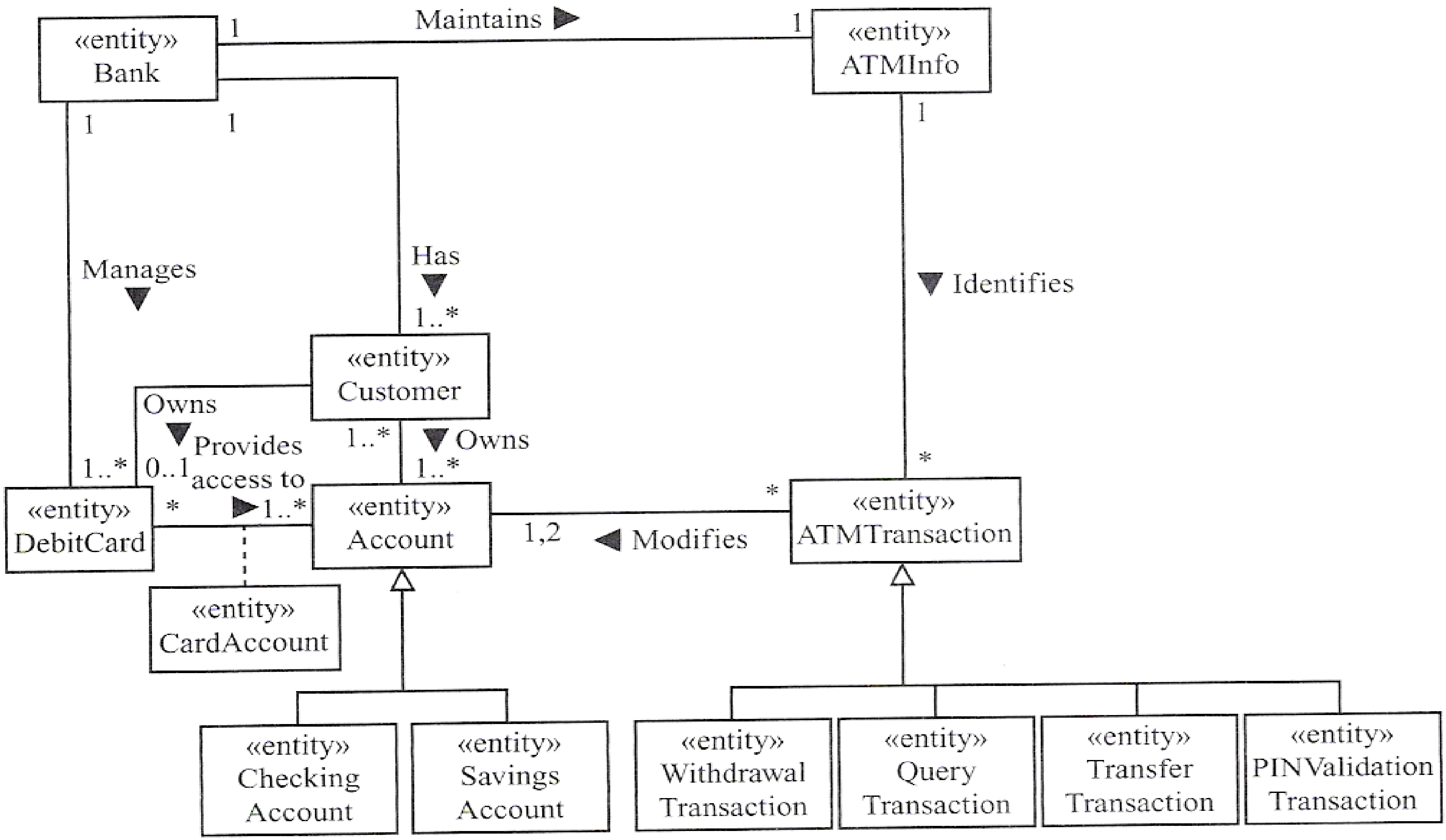
System context diagram

- The context diagram shows the interaction between:
  - the system to be designed (represented as a black box)
  - its environment (external users, external devices, etc.)
- defining precisely "what is the system" is important in order to differentiate between which classes/instances are included in the system and which are part of the environment
  - here, the system is a software application (does not include hardware devices or external users)
- Especially important for real-time and embedded systems, which interact with sensors, actuators, etc.



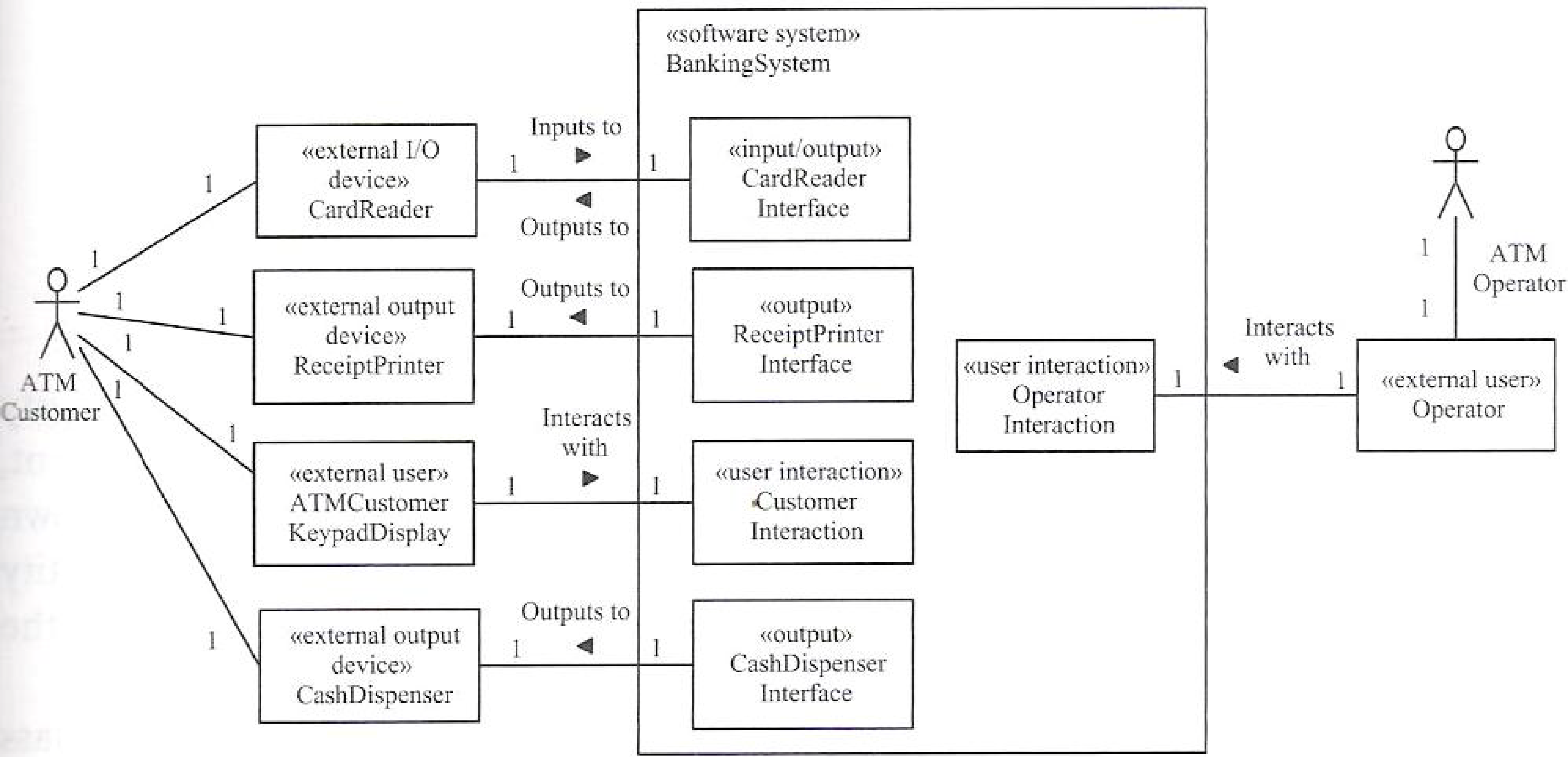
Entity classes for the problem domain

- The entity classes are data intensive classes that are encapsulating and storing information
  - may be persistent, in which case the entity object accesses a database (not shown in the analysis phase)
- the attributes are identified at this stage (not shown in the diagram)



External and boundary classes

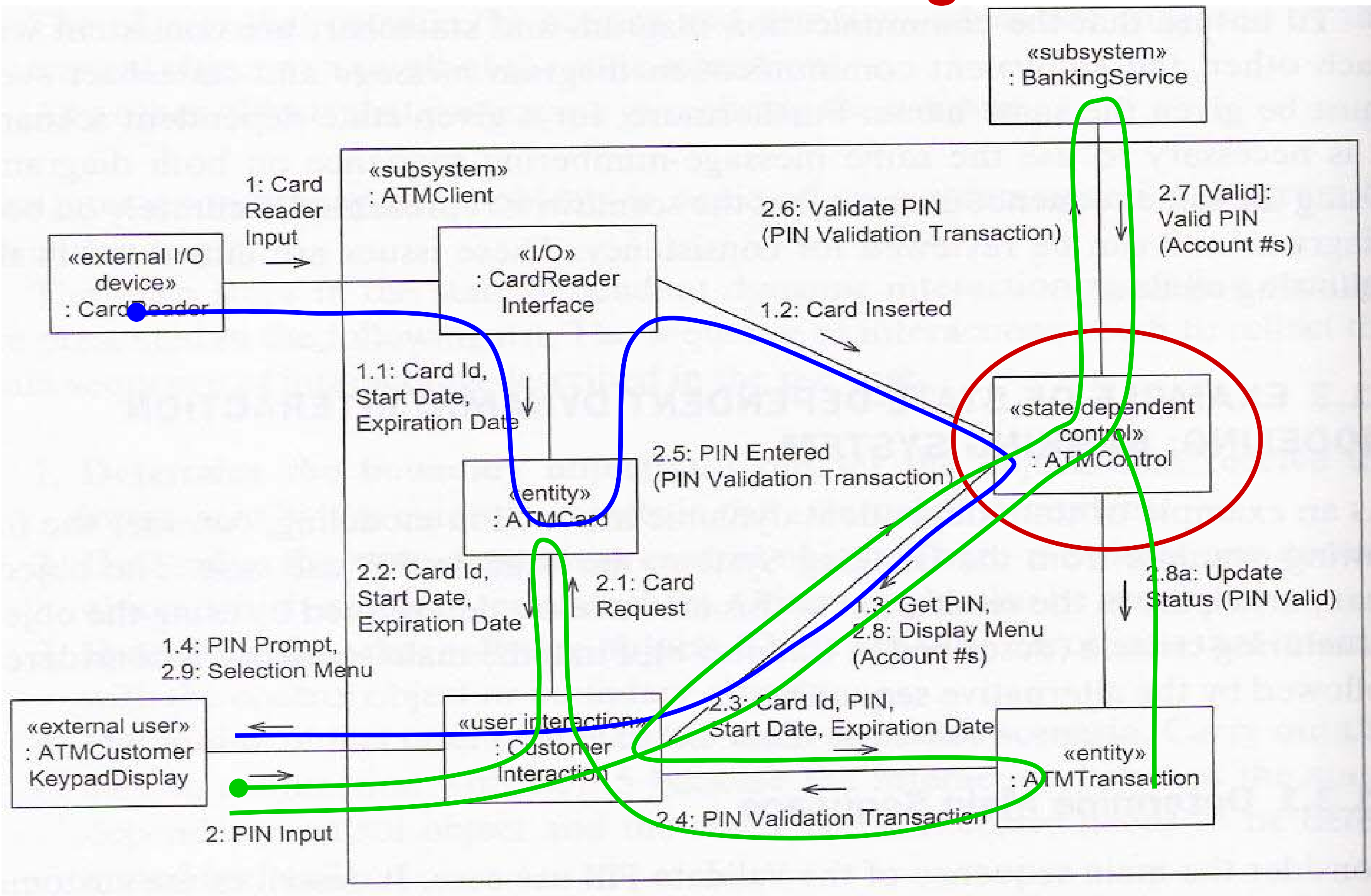
- **From the context diagram – add boundary objects inside the system that interact with the external objects identified previously**



Analysis phase: Dynamic model (i.e., behaviour)

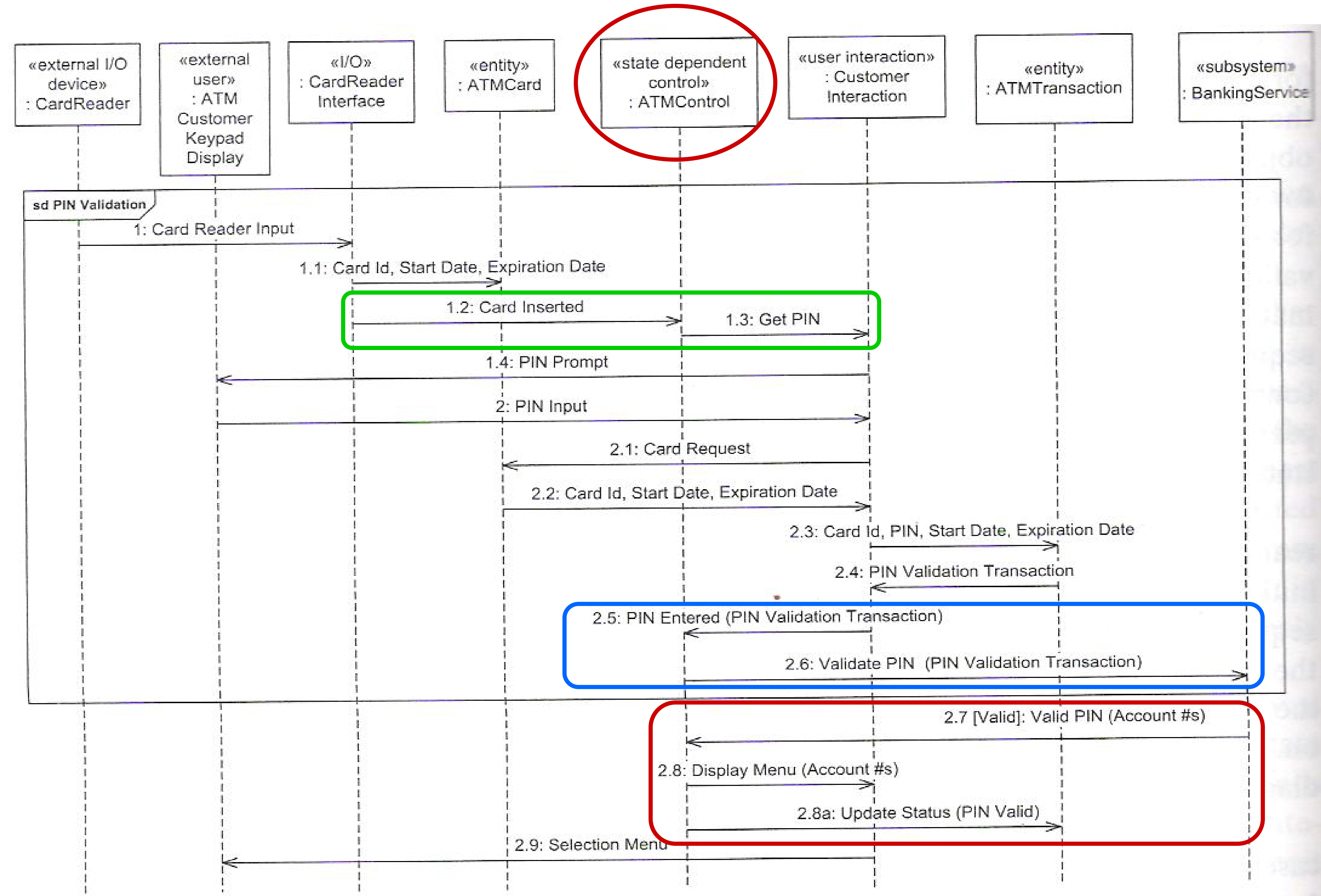
- **For each use case:**
  - identify participant objects (classes)
  - develop interaction diagrams
  - describe “happy path” and all alternatives
  - identify information passed between objects
  - develop a statechart for each state-dependent object

Validate PIN "happy path": communication diagram



PIN Validation Transaction = {transactionId, transactionType, cardId, PIN, startDate, expirationDate}

Validate PIN “happy path

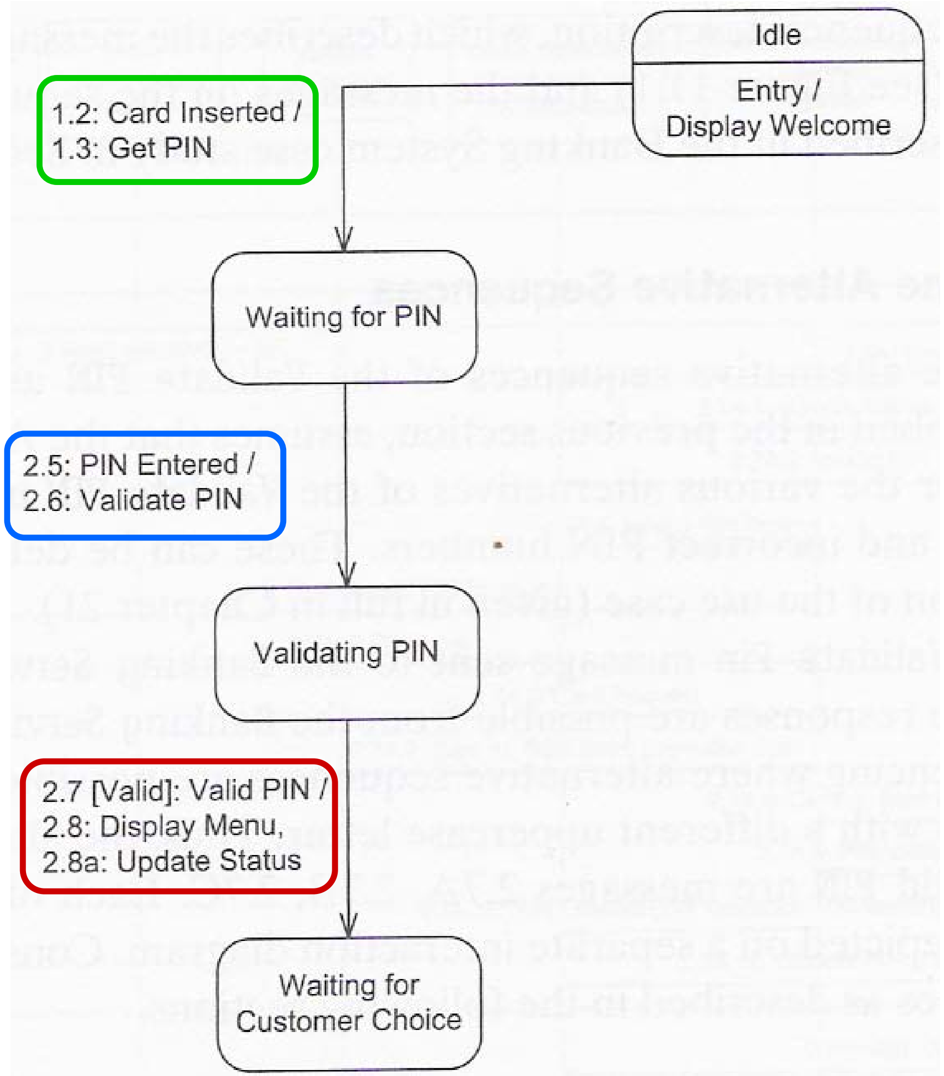


ATM Control: partial statechart for Validate PIN "happy path"

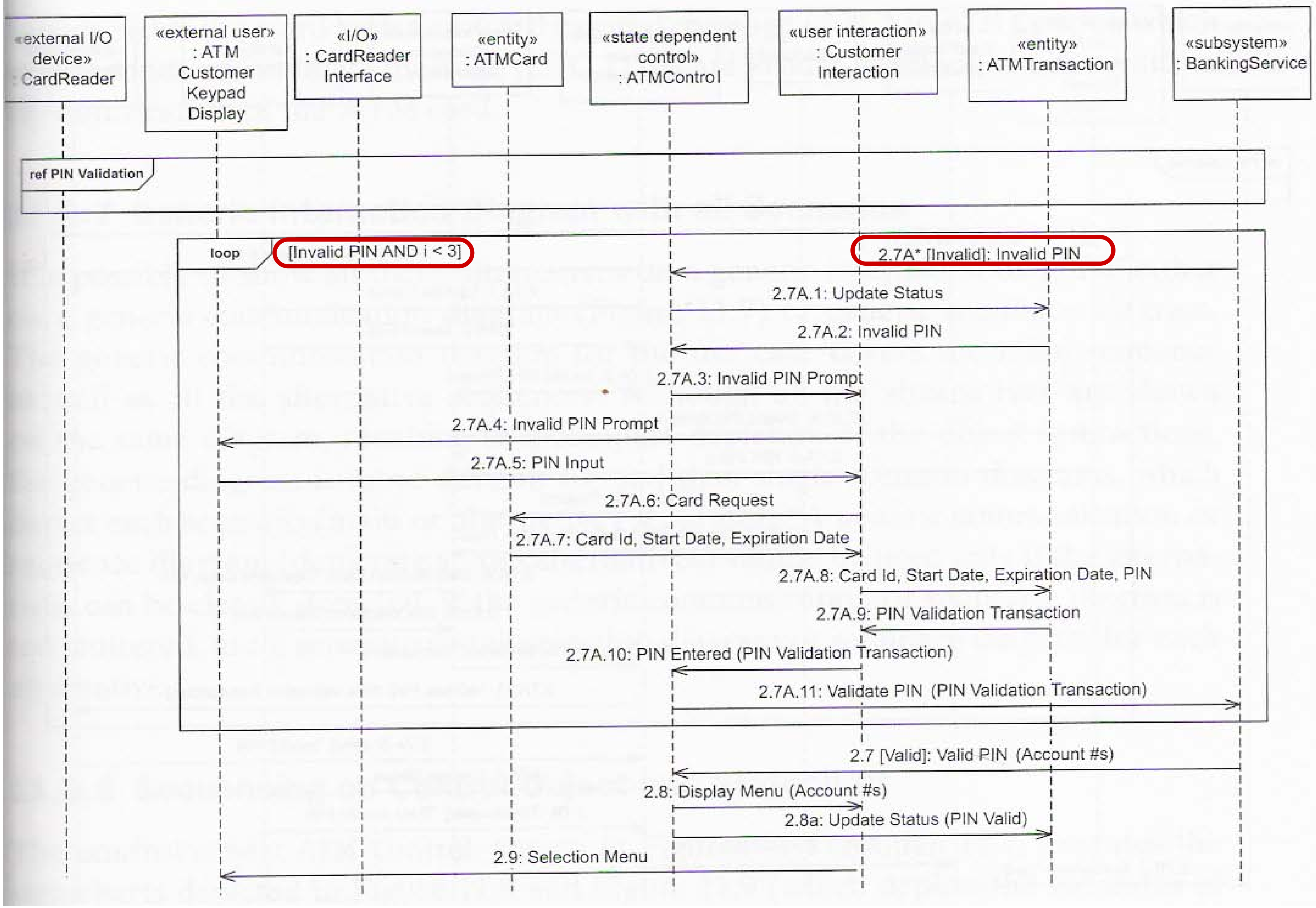
- **For each state-dependent object build a statechart corresponding to each use case realization**
  - **consider the interaction of ATM Control with other instances to build the ATM Control statechart**

Heuristics for building the statechart:

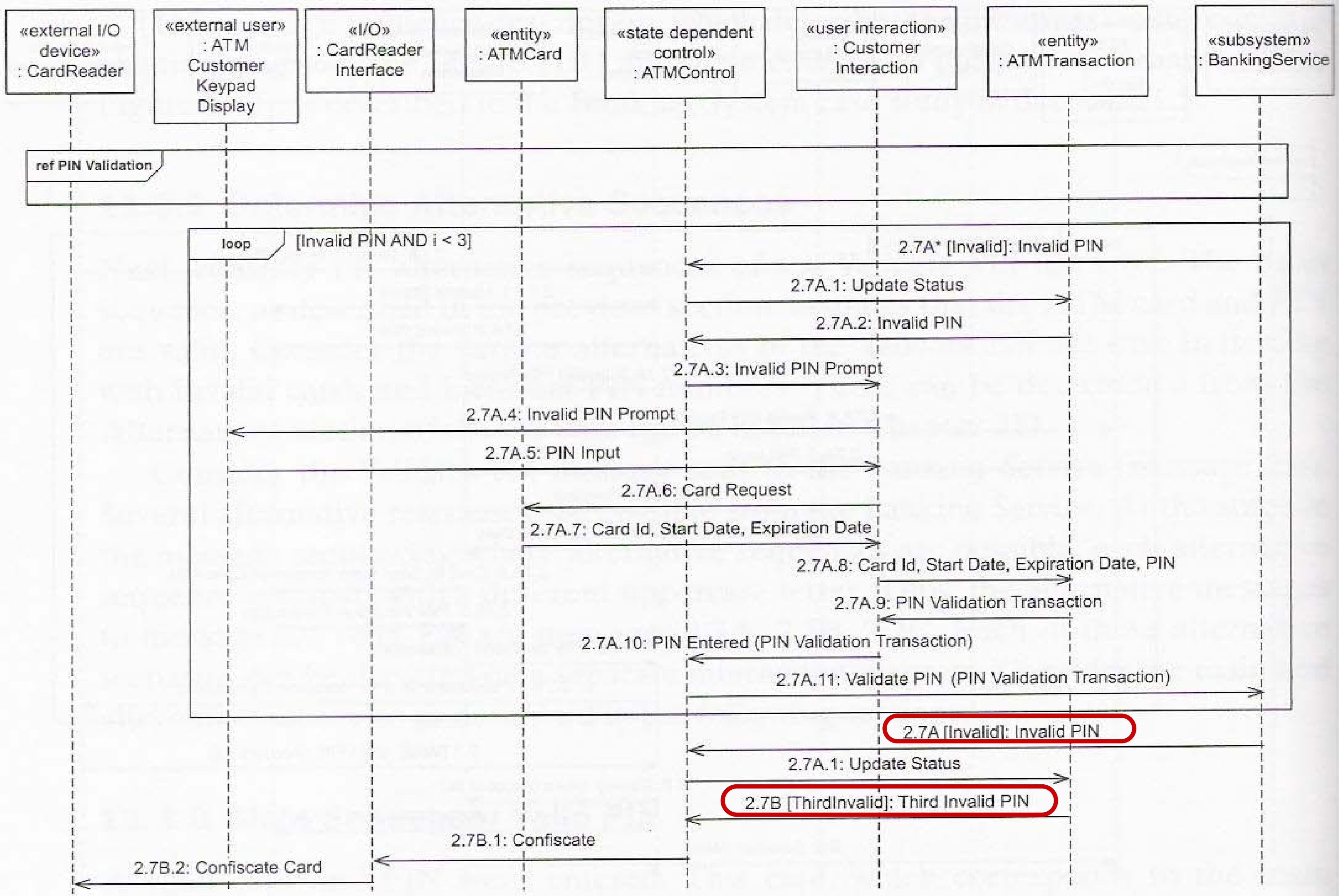
- **Start with the happy path**
  - **keep consistency between interaction diagram (ID) and statechart (SC):**
    - **incoming ID messages or signals correspond to SC triggers**
    - **outgoing ID messages or signals correspond to SC actions of sending messages**
    - **execution occurrences as effect of ID messages correspond to SC actions**
- **Continue with all the alternatives, adding new transitions, triggers, actions, and/or states to the statechart, as necessary.**



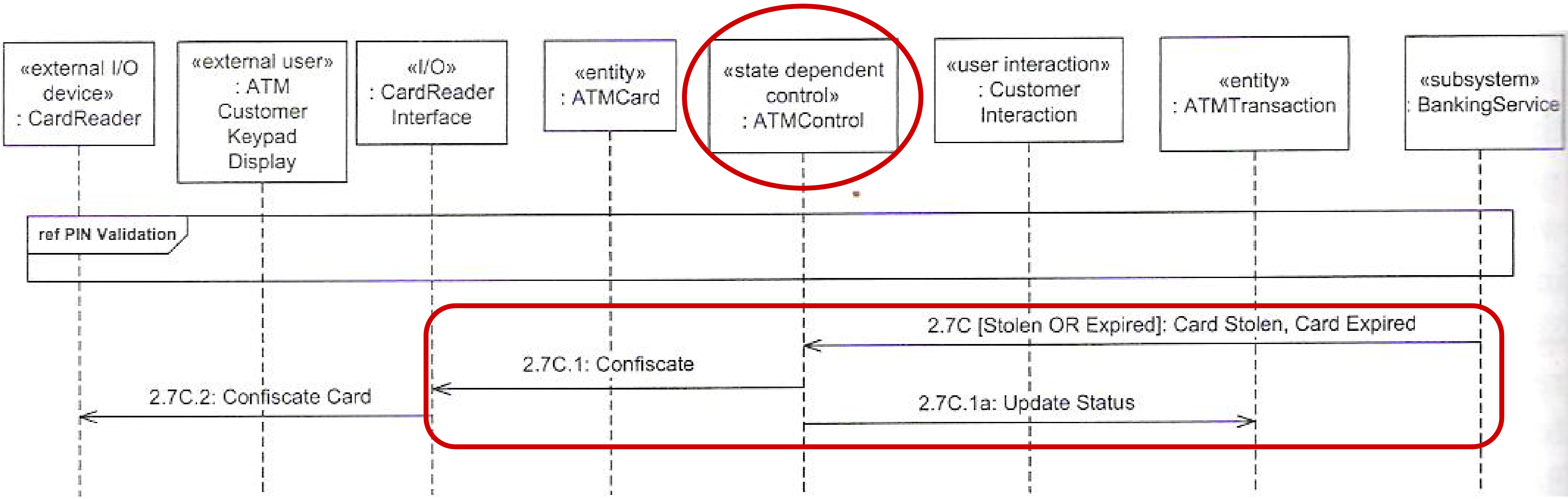
Validate PIN: Invalid PIN



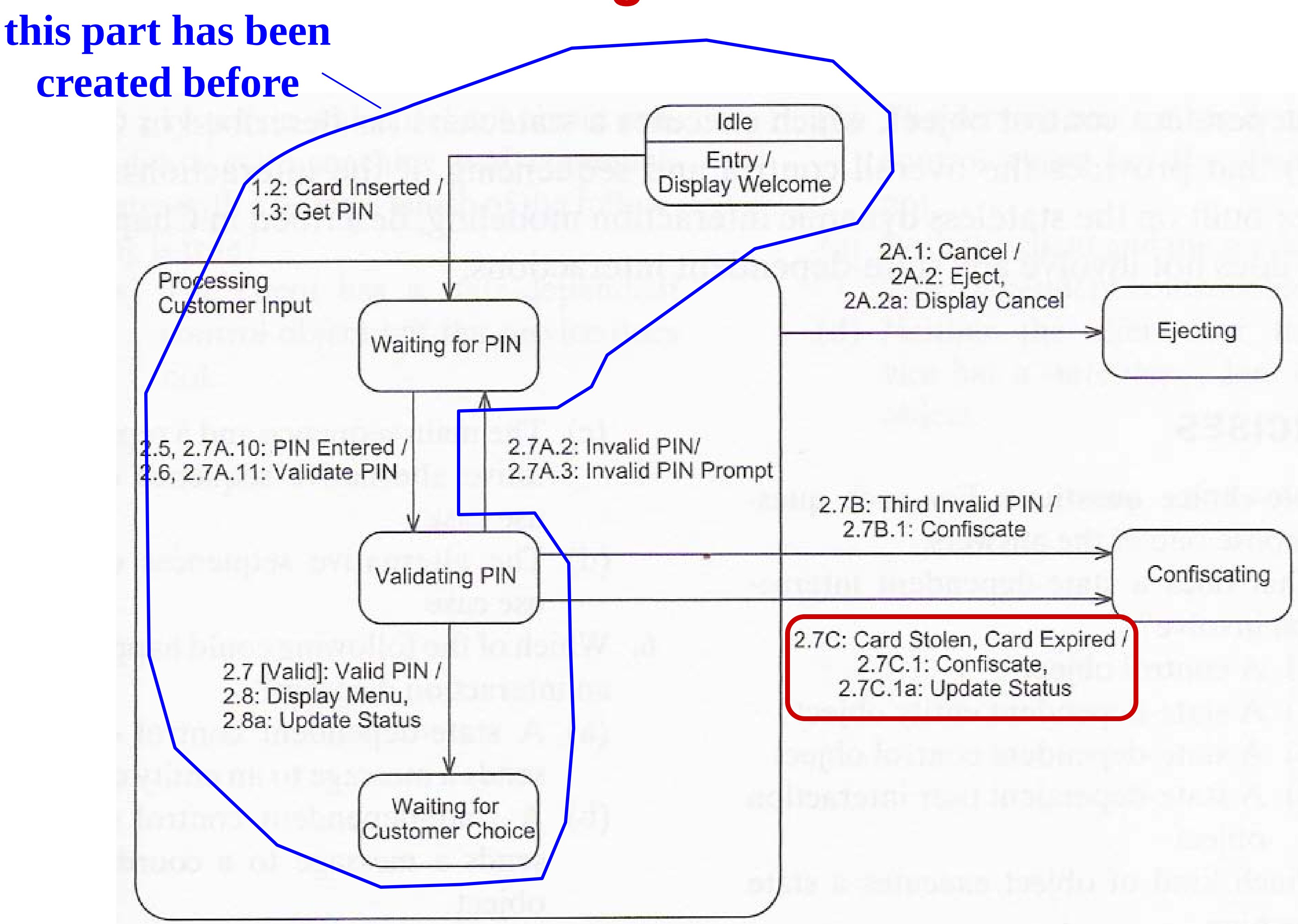
Validate PIN: Third Invalid PIN



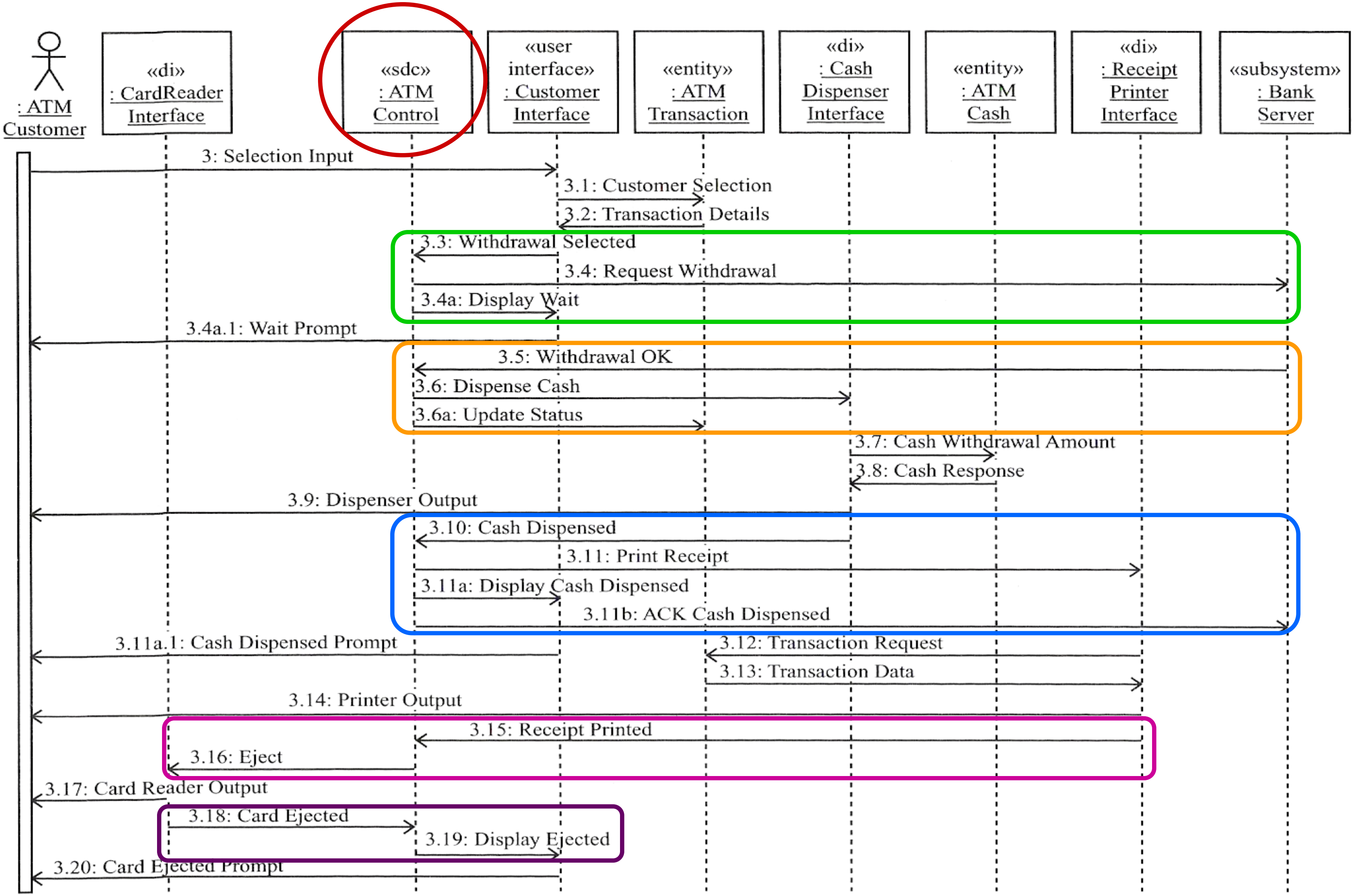
Validate PIN: card stolen or expired



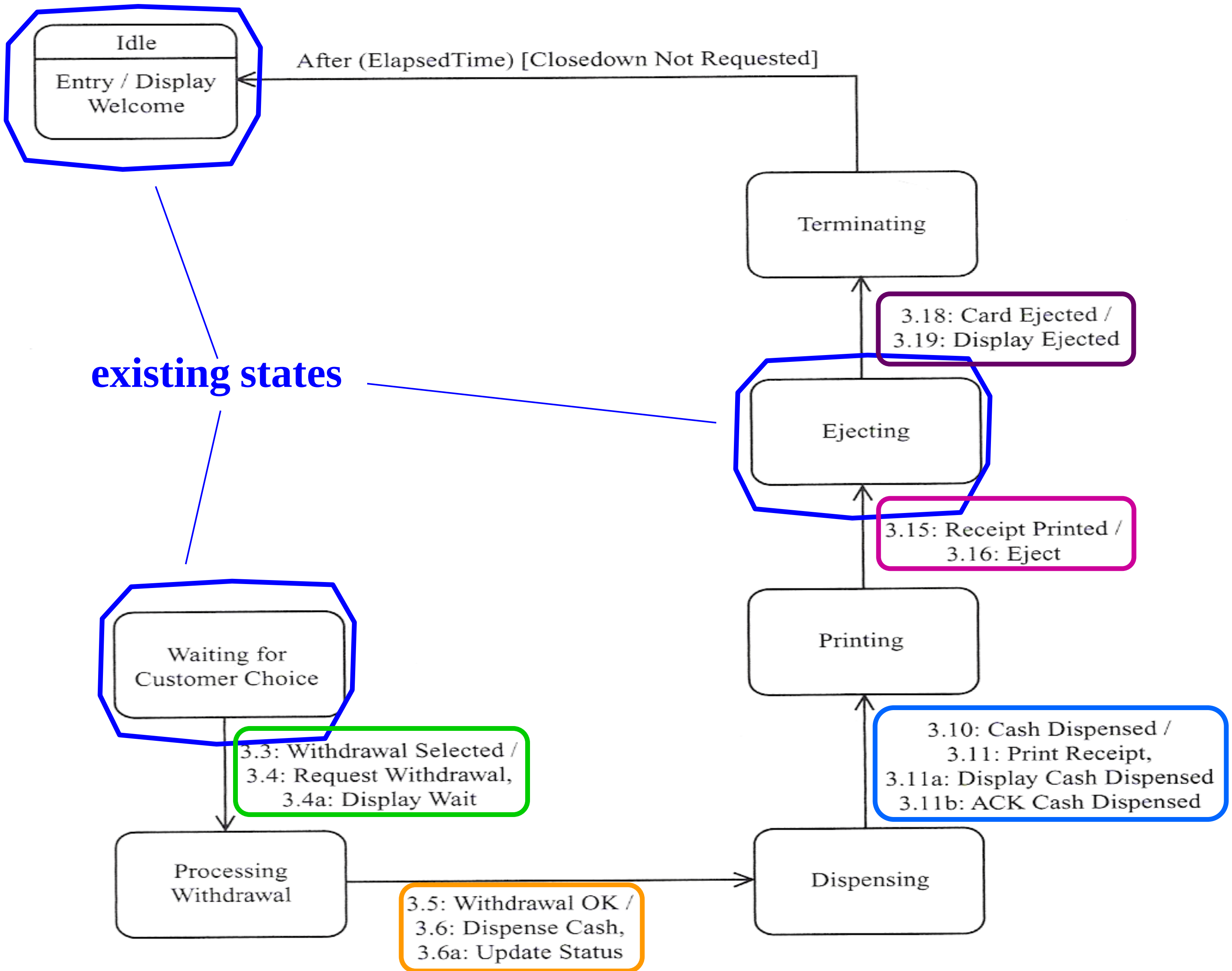
ATM Control: partial statechart for Validate PIN showing alternatives



Withdraw Funds

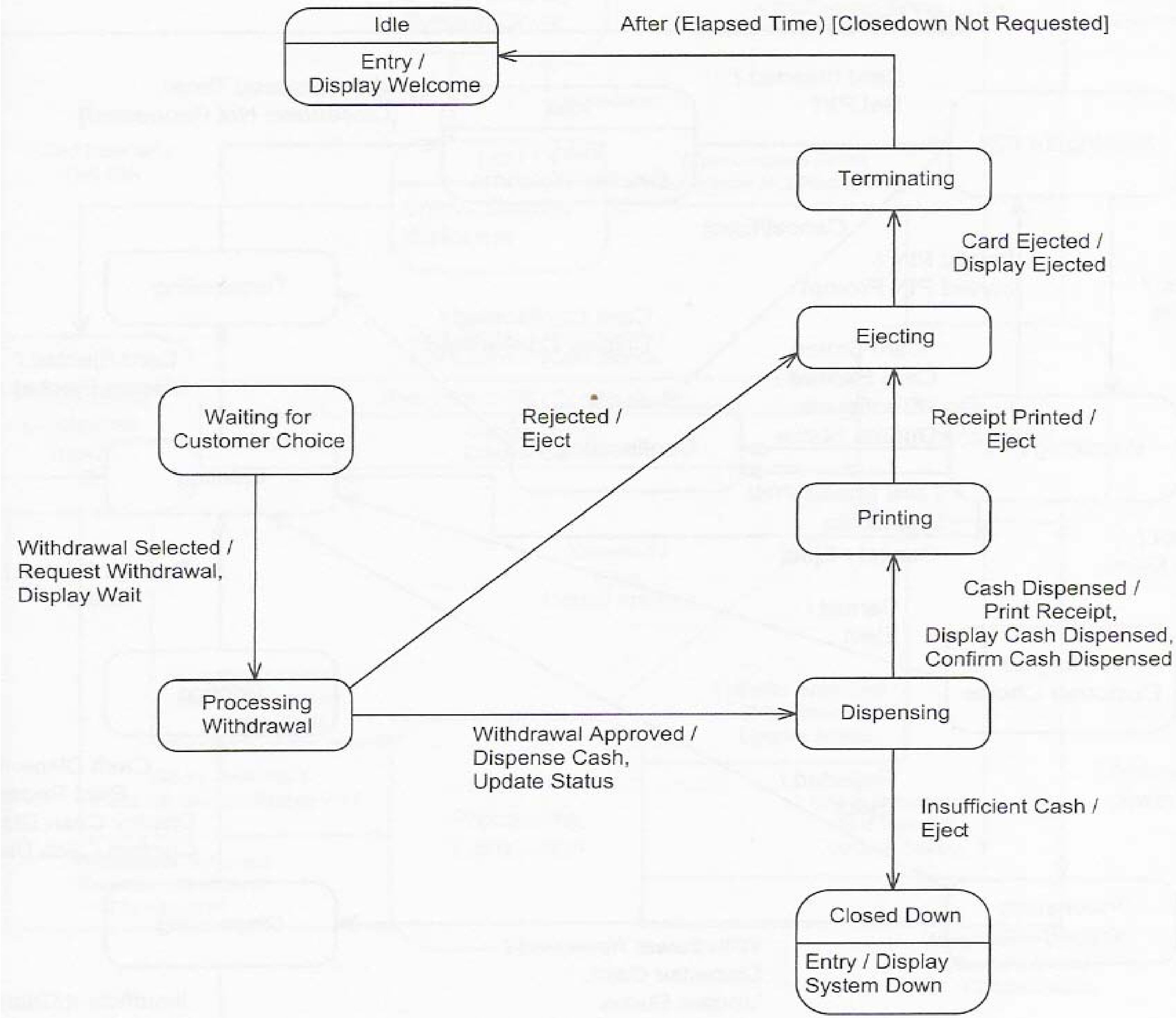


ATM Control: partial statechart for Withdraw Funds

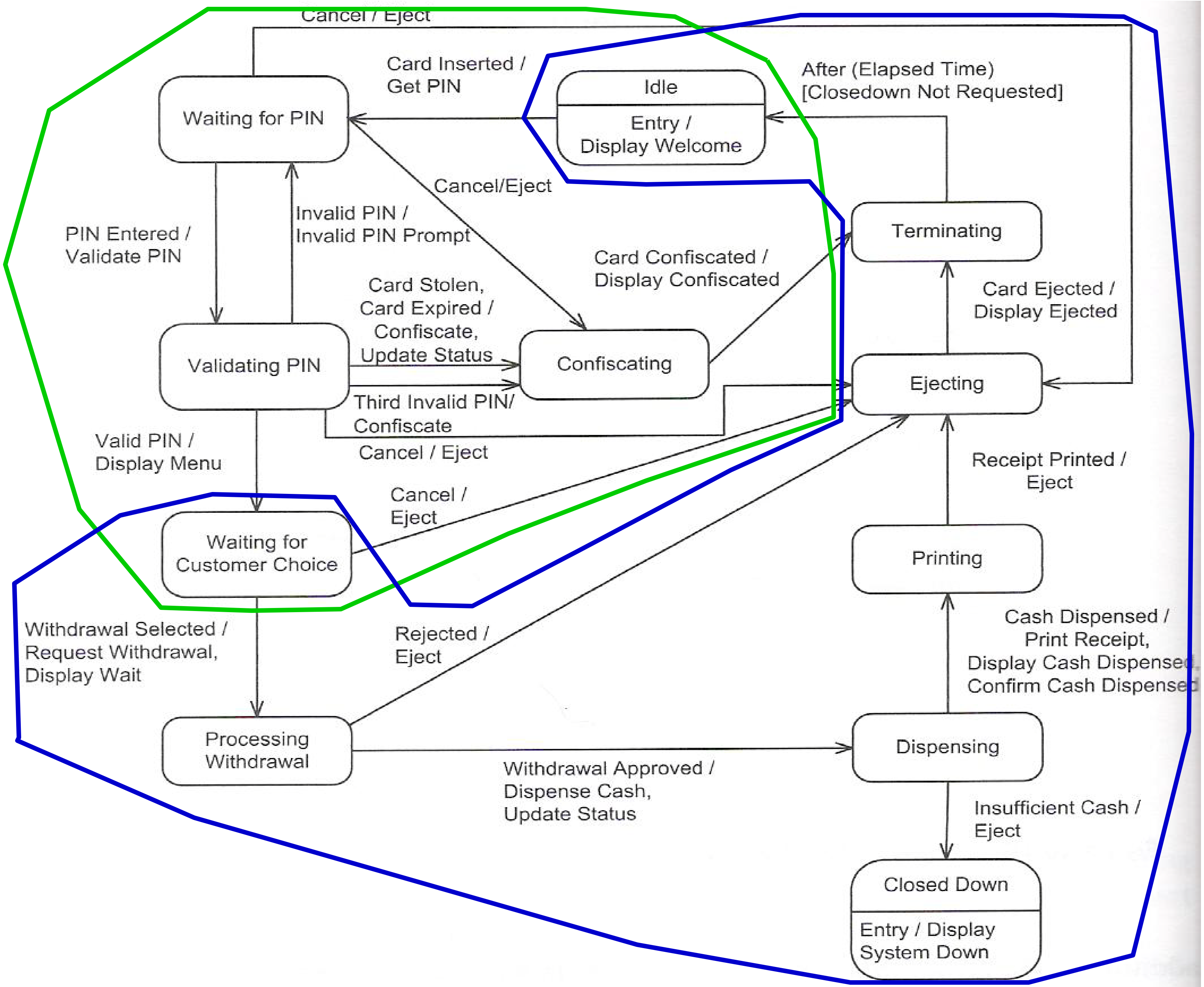


Carleton University
Canada's Capital University

Statechart for Withdraw Funds with alternatives

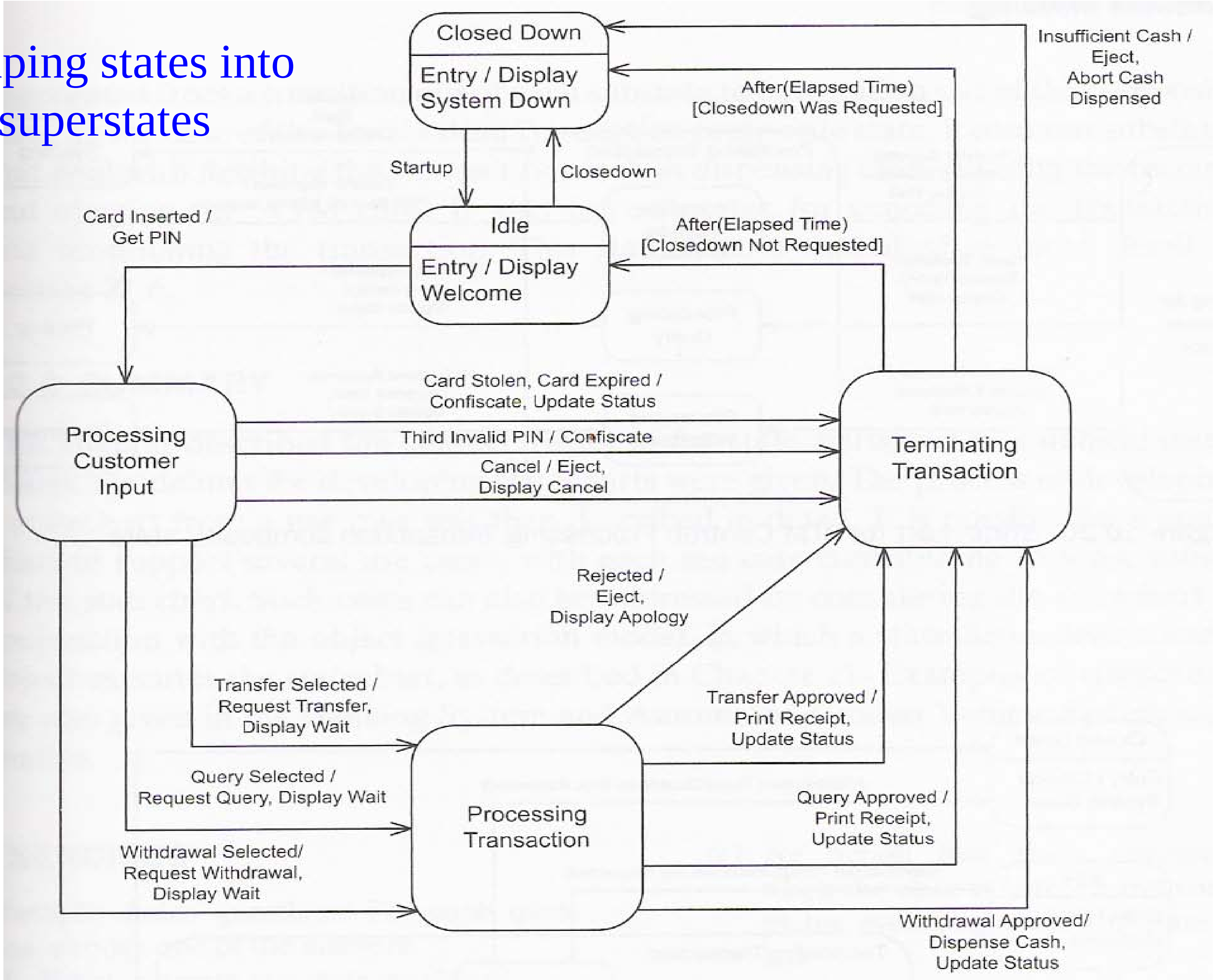


Integration: ValidatePIN and Withdraw Funds

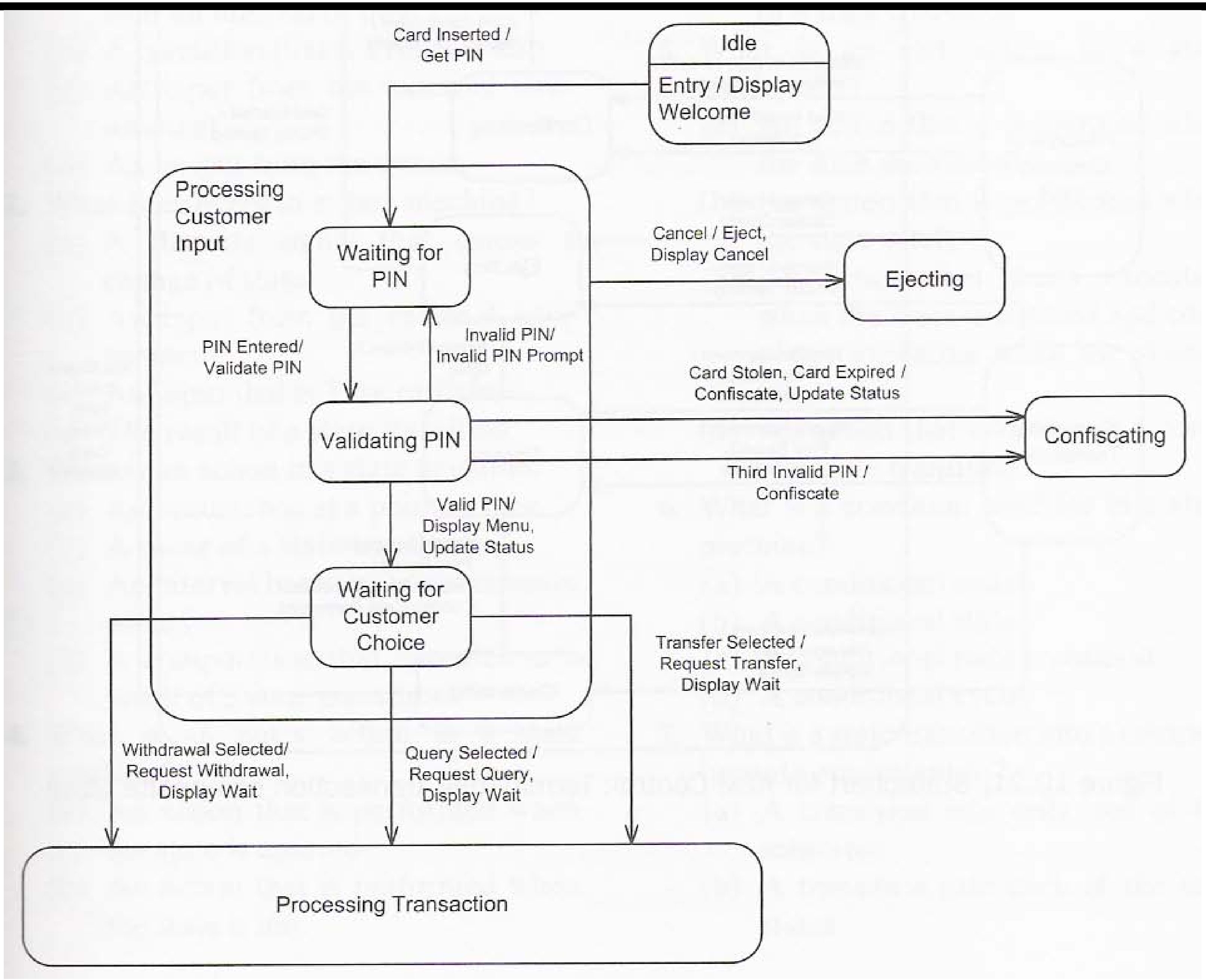


Hierarchical statechart for ATM Control: top level

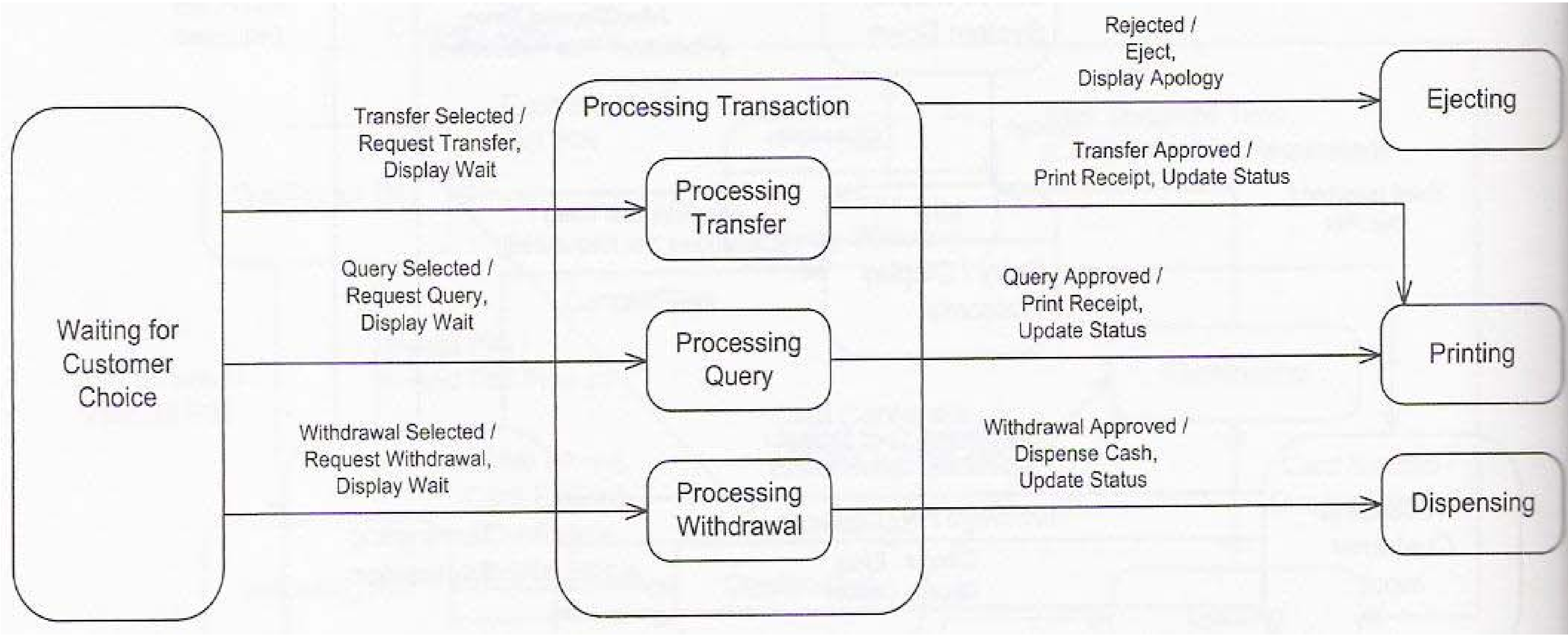
grouping states into superstates



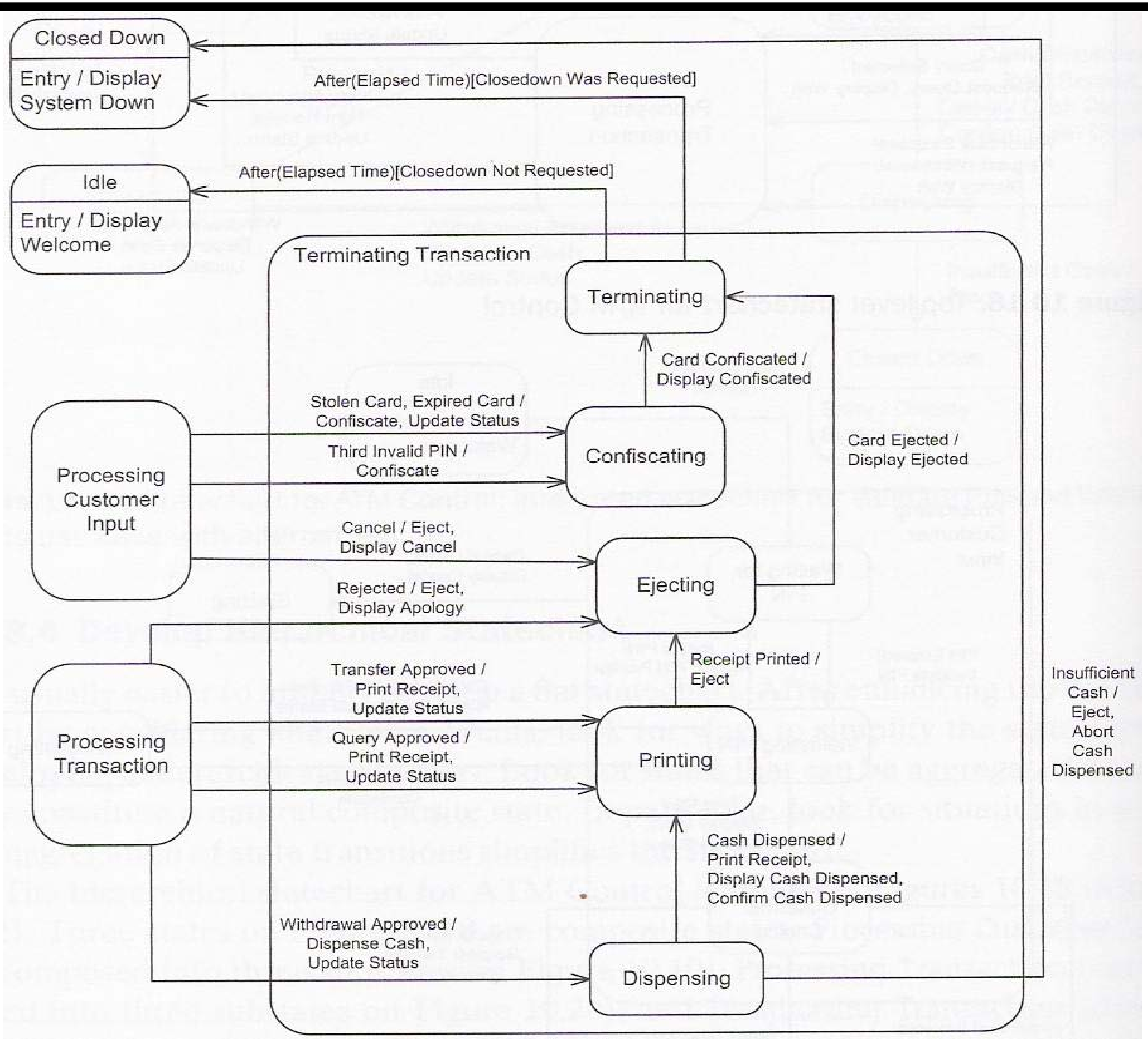
Processing Customer Input superstate



Processing Transaction superstate



Terminating Transaction superstate



System decomposition issues

- A system is structured into subsystems, which contain objects that are functionally dependent on each other (see example on next slide):
  - low coupling between subsystems
  - high coupling between objects in the same subsystem
  - a subsystem can be considered an aggregate or composite object that contains the objects that compose it
  - hierarchical decomposition can be used
- Separation of concerns between subsystems: each subsystem performs a major function which is relatively independent of other subsystems.
- Subsystems provide a larger-grained information hiding than objects.
- Guidelines for determining subsystems in the analysis phase
  - geographical subsystem structuring (Ex: ATM Banking System)
  - high coupling between objects in the same subsystem
    - try to group objects that participate in a use case into the same subsystem
    - objects participating in more us cases will be placed into one subsystem

Example of distributed software architecture

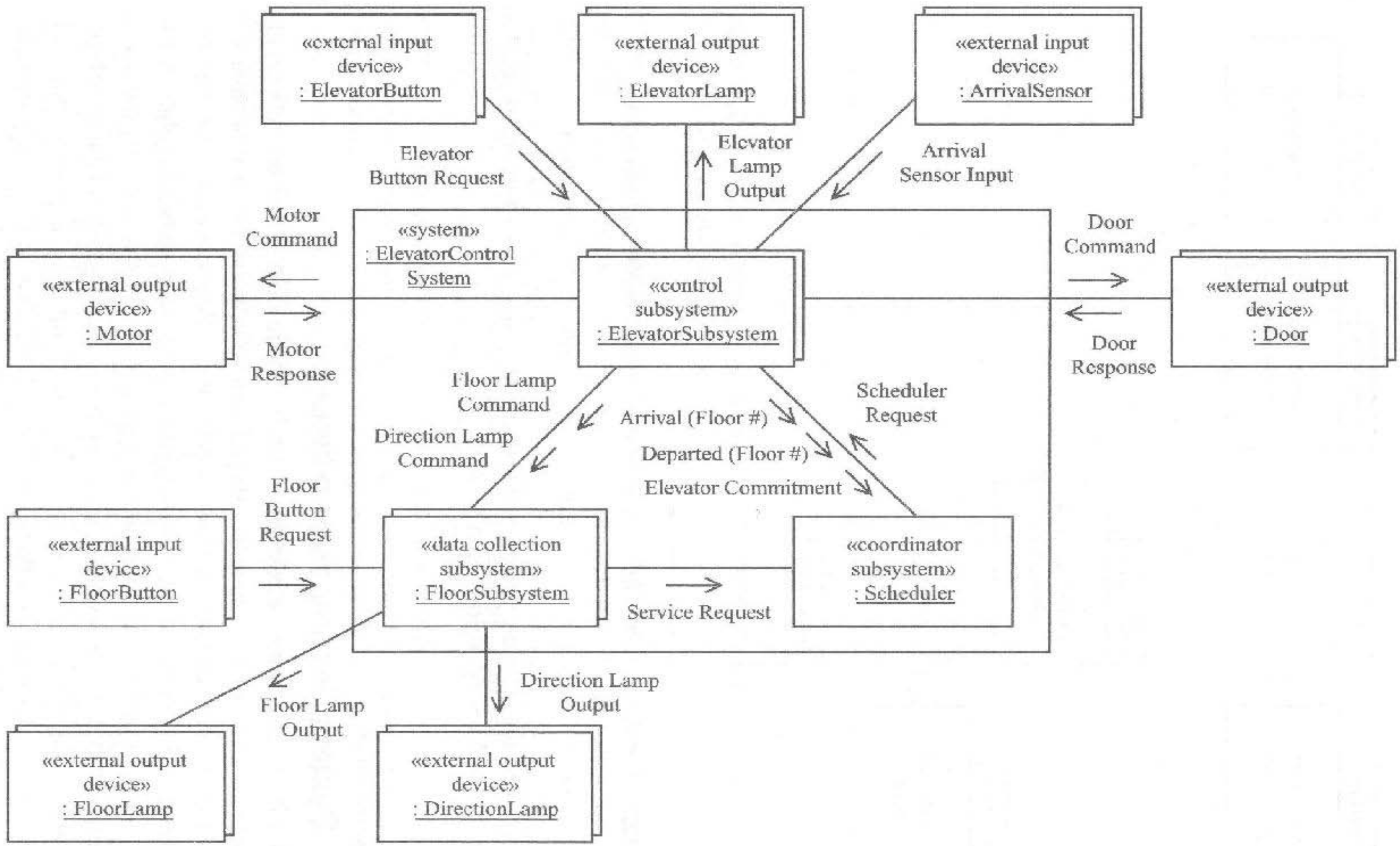


Figure 12.4 *Example of distributed software architecture: Elevator Control System*

Subsystem structuring criteria

- **Subsystems are likely to be application dependent. The kind of subsystems often needed in real-time systems is given below:**
  - **Control: controls a given aspect of the system or subsystem**
  - **Coordinator: in cases where there are more than one control subsystems, a coordinator may be necessary to coordinate them.**
  - **Data collection: collects data from the external environments. It may convert, store or reduce the data, usually in real time.**
  - **Data analysis: analyzes data and provides reports. As opposed to data collection, data analysis may be a non-real time activity.**
  - **Server**
  - **User interface**
  - **I/O subsystems**
  - **System services: file management, middleware, network communication.**
    - **usually not developed with the application, but the designer has to recognize their existence and use them.**