Karlstad University
Department of Information Systems

Adapted for a textbook by Blaha M. and Rumbaugh J.

Object Oriented Modeling and Design

Pearson Prentice Hall, 2005

Development Process: Analysis and Design

Remigijus GUSTAS

Phone: +46-54 700 17 65

E-mail: Remigijus.Gustas@kau.se

http://www.cs.kau.se/~gustas/

OO Development Process:

- ✓ System Conception (requirements elicitation).
- ✓ System Analysis
  - Domain analysis
  - Application analysis
- ✓ System Design.
- ✓ Implementation.
- ✓ Testing.
- ✓ Training.
- ✓ Deployment.
- ✓ Maintenance.

System Conception

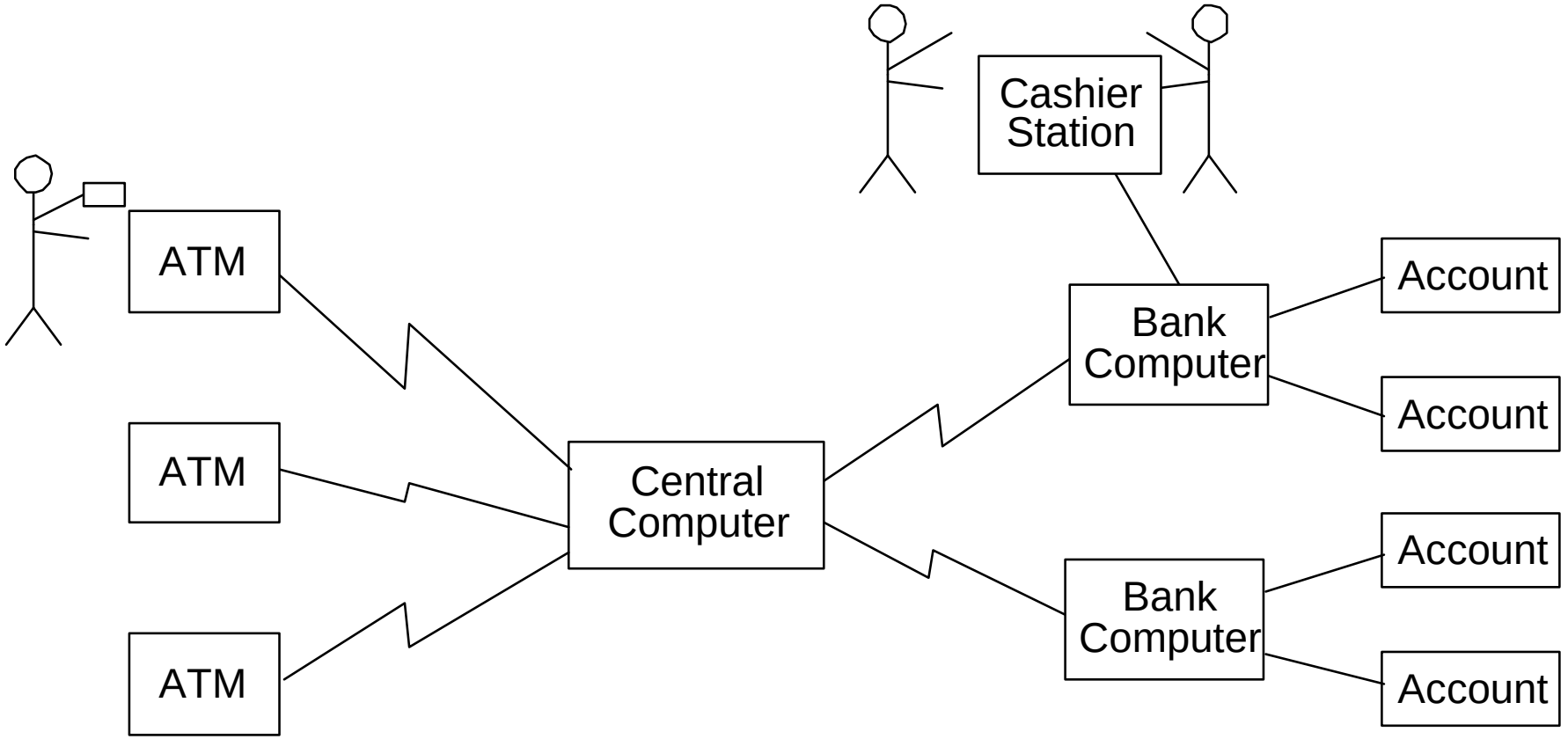
- Requirement analysis phase
  - Requirements describe how system behaves from the user's point of view.
  - True customer requirements should be separated from design decisions.
  - Solution should be deferred until a problem is fully understood.
- Requirements statements are typically **ambiguous, incomplete and inconsistent**.
- Some of requirements are plain **wrong**, some impose **unreasonable implementation cost** or may **create new problems**, if implemented.

Problem (Requirements) Statement

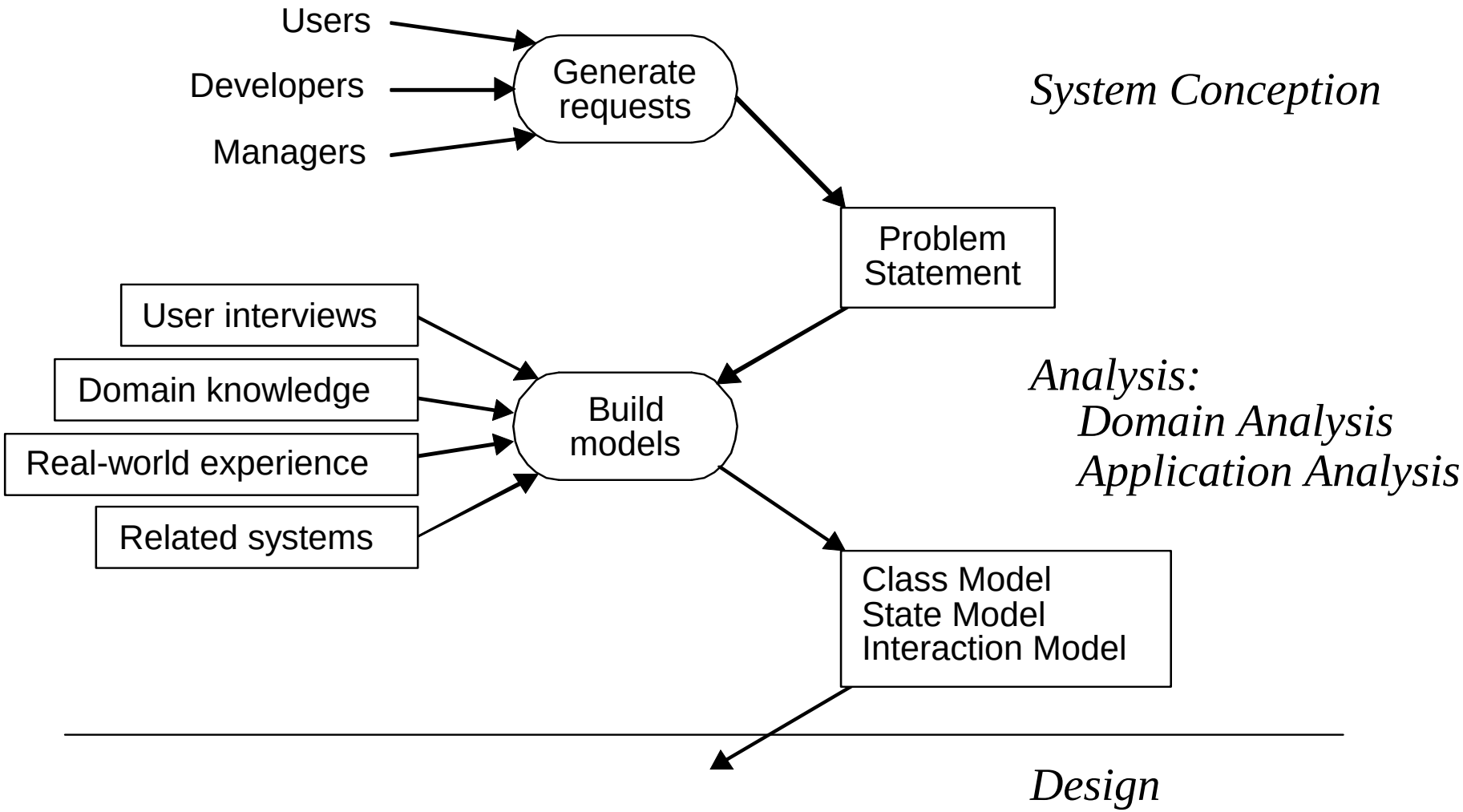
- ✓ Problem (Requirements) Statement should define what is to be done and not how it is to be designed or to be implemented

Requirements Statement	Design	Implementation
• Problem scope • What is needed • Application context • Assumptions • Performance needs	• General approach • Algorithms • Data structures • Architecture • Optimizations • Capacity planning	• Platforms • Hardware specs • Software libraries • Interface standards

System Conception in a graphical form



Domain Analysis

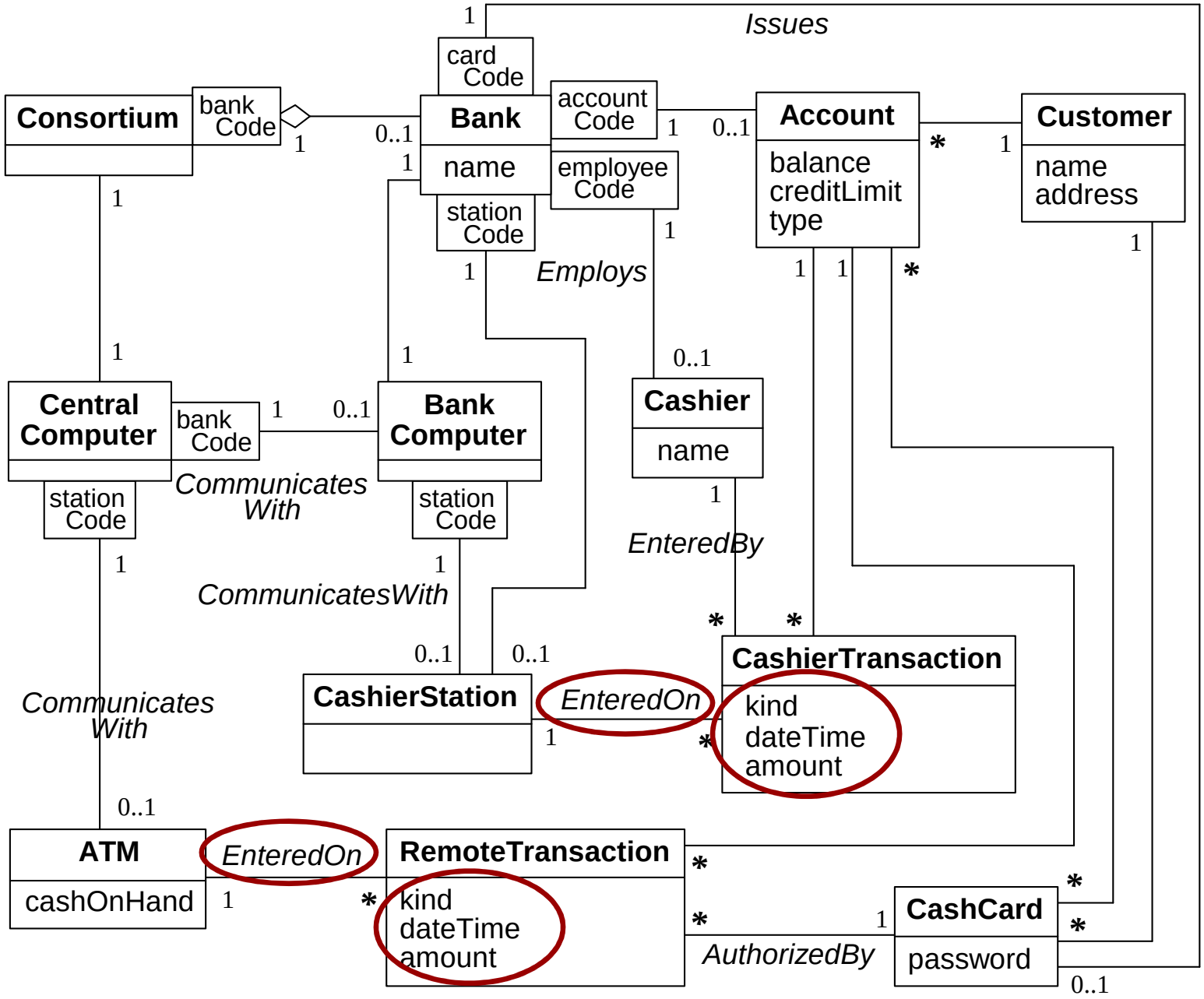


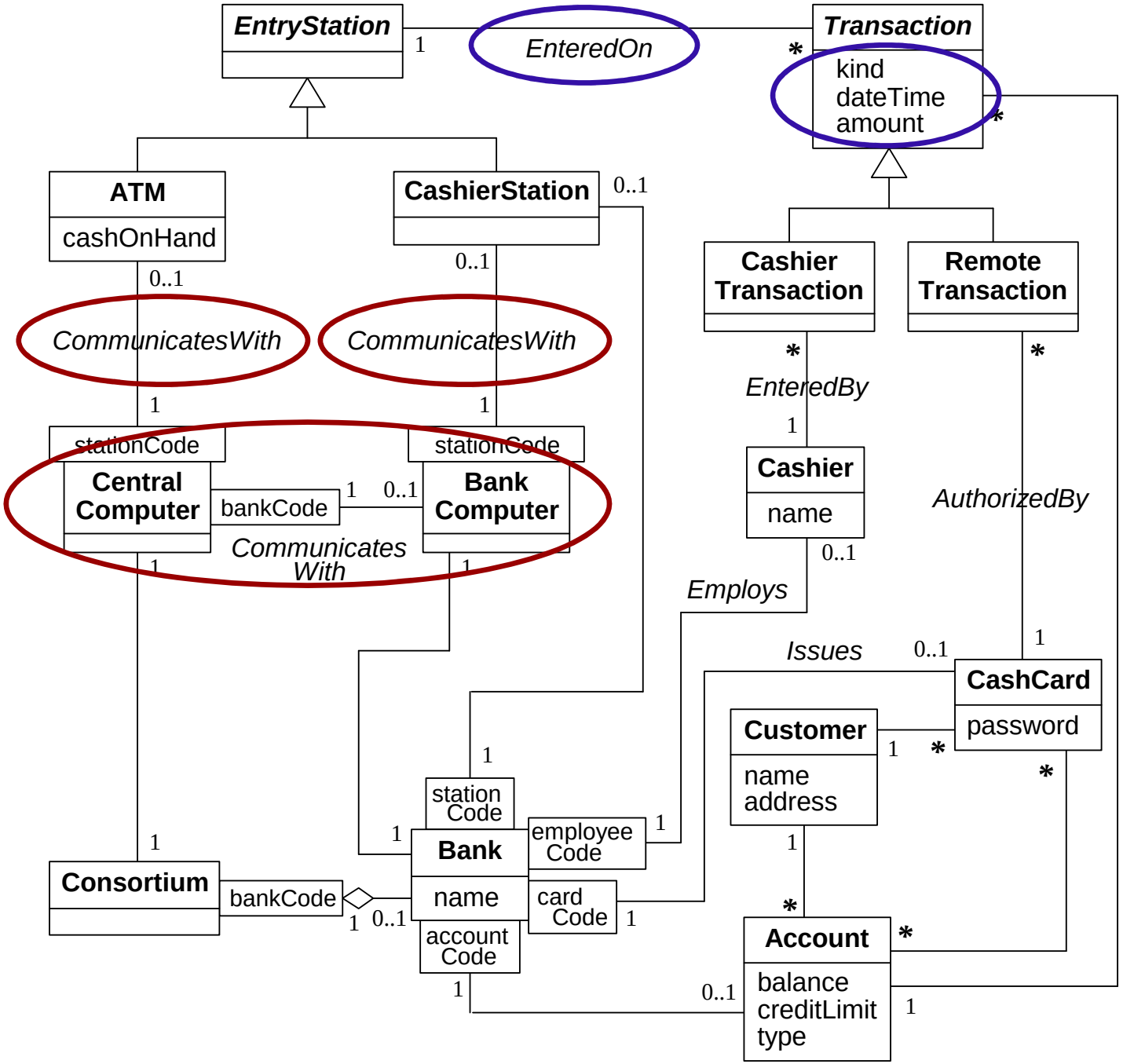
Keeping the Right Concepts in Domain Analysis Phase

- ✓ Redundant (synonyms) classes, operations and attributes (the most descriptive name should be chosen).
- ✓ Irrelevant concepts (if a class, operation or attribute has nothing to do with the problem, it should be eliminated.
- ✓ Vague classes (some classes may be too broad in scope, they should be specific).
- ✓ Attributes, roles and operations that have features of its own should be interpreted as classes.
- ✓ Derived and implementation classes, associations and attributes should be eliminated.
- ✓ Reconsider all boolean attributes.

Refining with Inheritance

- ✓ Inheritance can be added by generalizing existing classes into a superclass or by specializing a class into subclasses (based on taxonomic relations).
- ✓ Inheritance can be discovered by searching classes with similar attributes, associations and operations.
- ✓ Enumeration (values of some attribute) can be replaced by generalization (or Power Type).
- ✓ Multiple inheritance introduced, if it is absolutely necessary (increases complexity).
- ✓ Each attribute and association should be assigned to the most general class for which it is appropriate.
  - When the same name of an attribute or association appears more than once, the associated classes can be generalized.


Issues
card Code
Consortium
bank Code
Bank
account Code
Account
Customer
name
employee Code
balance
creditLimit
type
name
address
station Code
Employs
Central Computer
bank Code
Communicates With
Bank Computer
station Code
station Code
Cashier
name
EnteredBy
CommunicatesWith
CashierTransaction
kind
dateTime
amount
CashierStation
EnteredOn
Communicates With
ATM
EnteredOn
cashOnHand
RemoteTransaction
kind
dateTime
amount
AuthorizedBy
CashCard
password

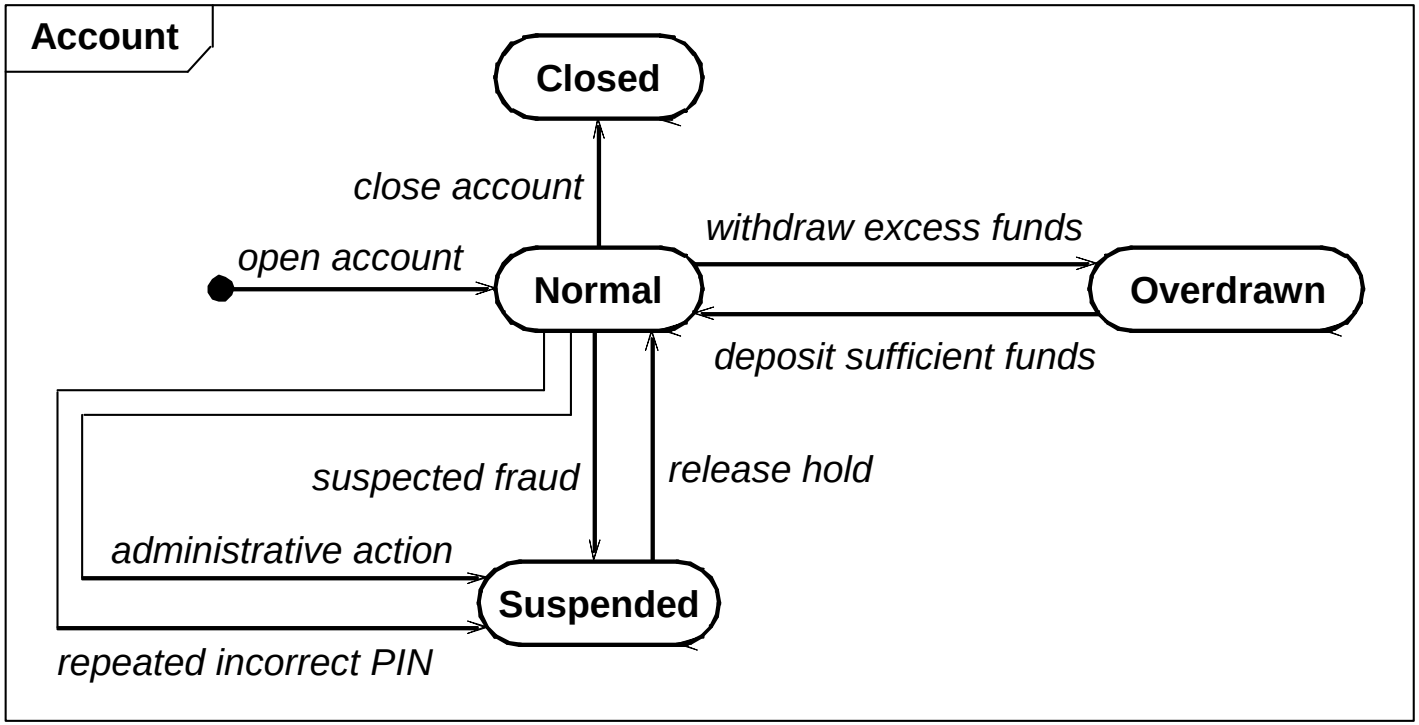

EntryStation
1
EnteredOn
*
Transaction
kind
dateTime
amount
*
ATM
cashOnHand
0..1
CashierStation
0..1
0..1
Cashier Transaction
Remote Transaction
CommunicatesWith
CommunicatesWith
*
*
EnteredBy
1
1
1
stationCode
stationCode
Central Computer
bankCode
1
0..1
Bank Computer
Communicates With
1
1
Cashier
name
0..1
AuthorizedBy
Employs
Issues
0..1
1
CashCard
password
Customer
name
address
1
*
*
1
station Code
1
Bank
name
account Code
employee Code
1
card Code
1
Consortium
bankCode
1
0..1
1
*
Account
balance
creditLimit
type
*
0..1
1

Domain Analysis: Building State Model

- ✓ Identifying classes with states
- ✓ Finding states
  - States should be based on qualitative differences
- ✓ Identifying events
  - In many cases an event can be as completion of another activity
- ✓ Defining state diagrams
  - If an event has different effects in different states, add a transition for each state
- ✓ Evaluating state diagrams

Evaluating state diagrams

- Are all states connected?
- Is there a path from the initial state to the final state?
- Are there any dead states that terminate lifecycle?



Domain Analysis Summary

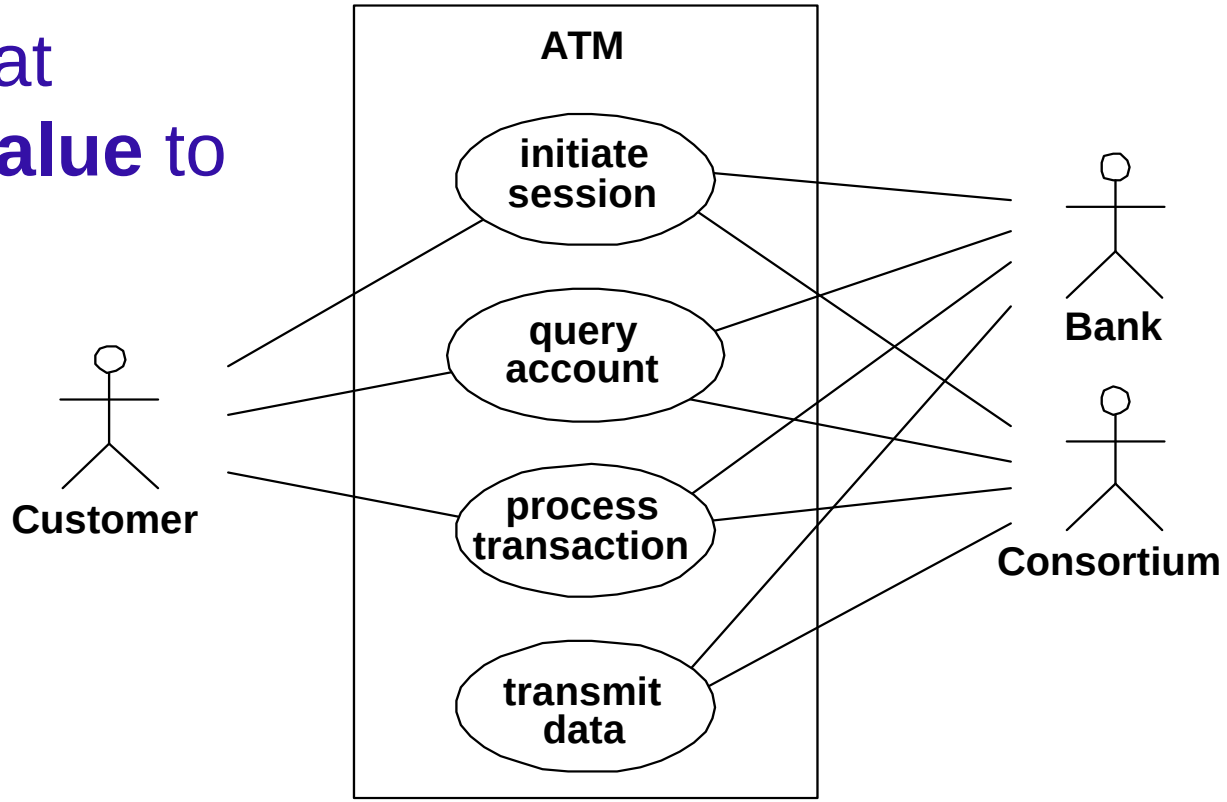
- ✓ OO domain analysis model contain class models, often state models, but seldom has an interaction model.
- ✓ The goal is to analyze a problem without introducing bias for implementation.
- ✓ Business experts should validate the analysis model.
- ✓ Analysis models can be used as an effective means of communication among business experts and system design experts.

Application Analysis: Building Interaction Model

- ✓ Determine system boundary.
- ✓ Identify actors and use cases.
- ✓ Determine initial and final events in each use case.
- ✓ Define scenarios for normal course of events.
- ✓ Define alternative scenarios.
- ✓ Identify external events.
- ✓ Prepare activity diagrams for use cases.
- ✓ Identify dependencies among actors and use cases.
- ✓ Consistency checking against the domain model.

Determine system boundary; Identify actors and use cases

Use case should represent a **service** that provides **value** to an actor



Determine initial and final events in each use case

- ✓ Use cases partition system functionality, but they do not show the behavior clearly.
- ✓ The initial event is a request for service
  - It is necessary to determine by which actor an event is initiated
- ✓ The final events determine the scope of the use case by defining when it terminates.

Use case: Query account

Initial event: Request for account data

Final event: Delivery of account data to the customer

Define scenarios

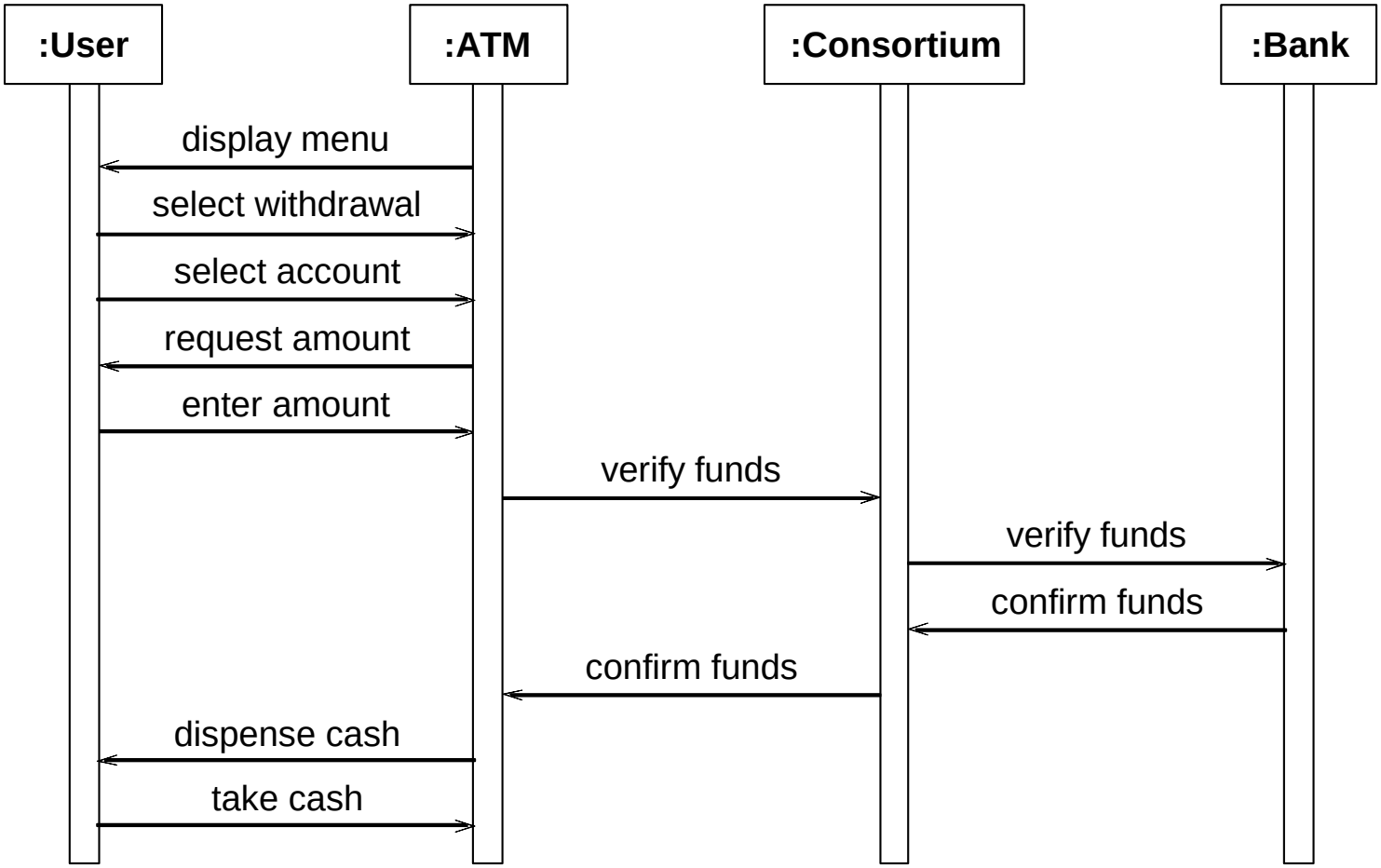
- ✓ Define scenarios for normal course of events.
- ✓ Define alternative scenarios.

Use case: Query account

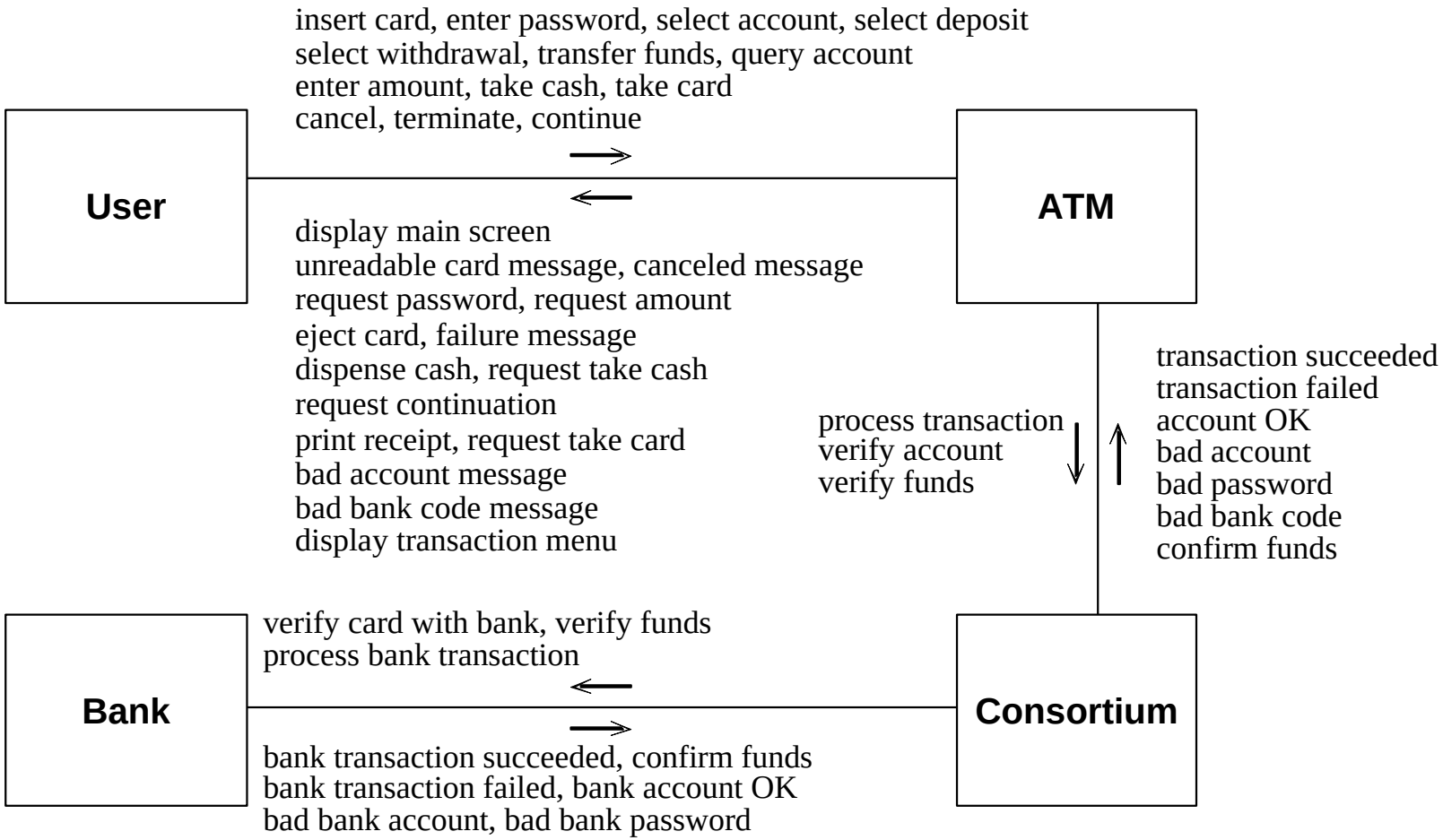
The ATM displays a menu of accounts and commands
The user chooses to query an account
The ATM contacts the consortium and bank which return the data
The ATM displays account data for the user
The ATM displays a menu of accounts and commands

Identify External Events:

Prepare a sequence diagram for each scenario

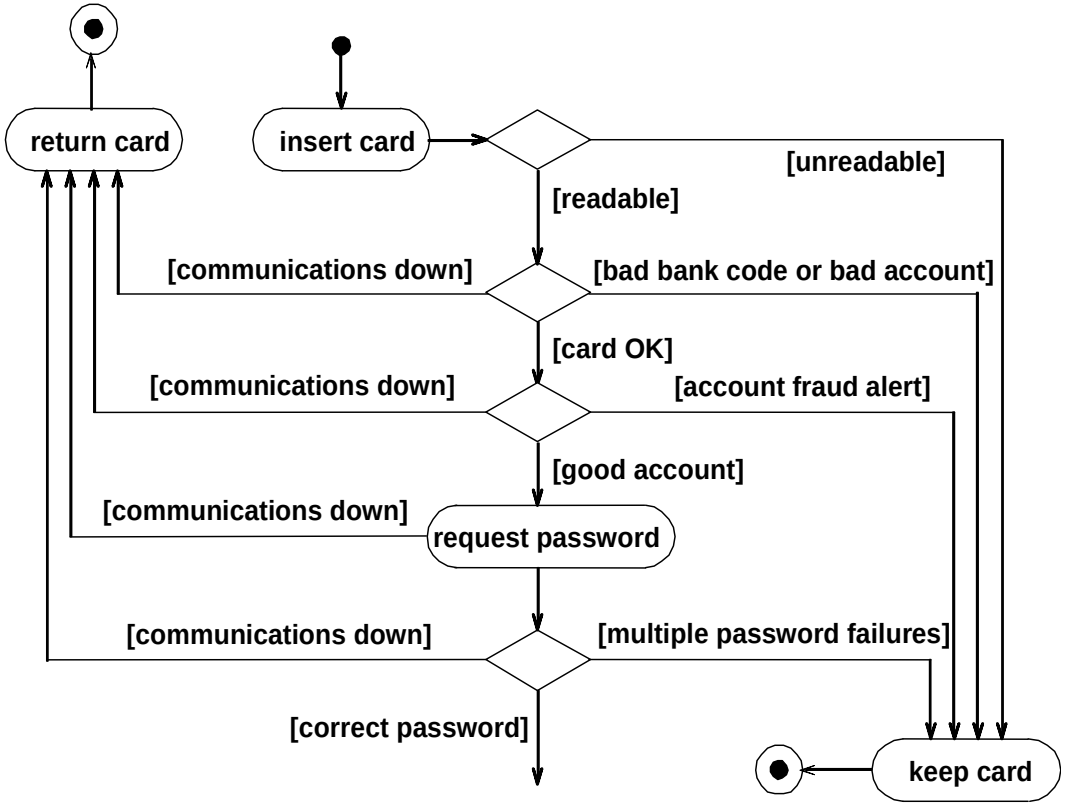


Identify External Events: Diagram with the Summary of Events



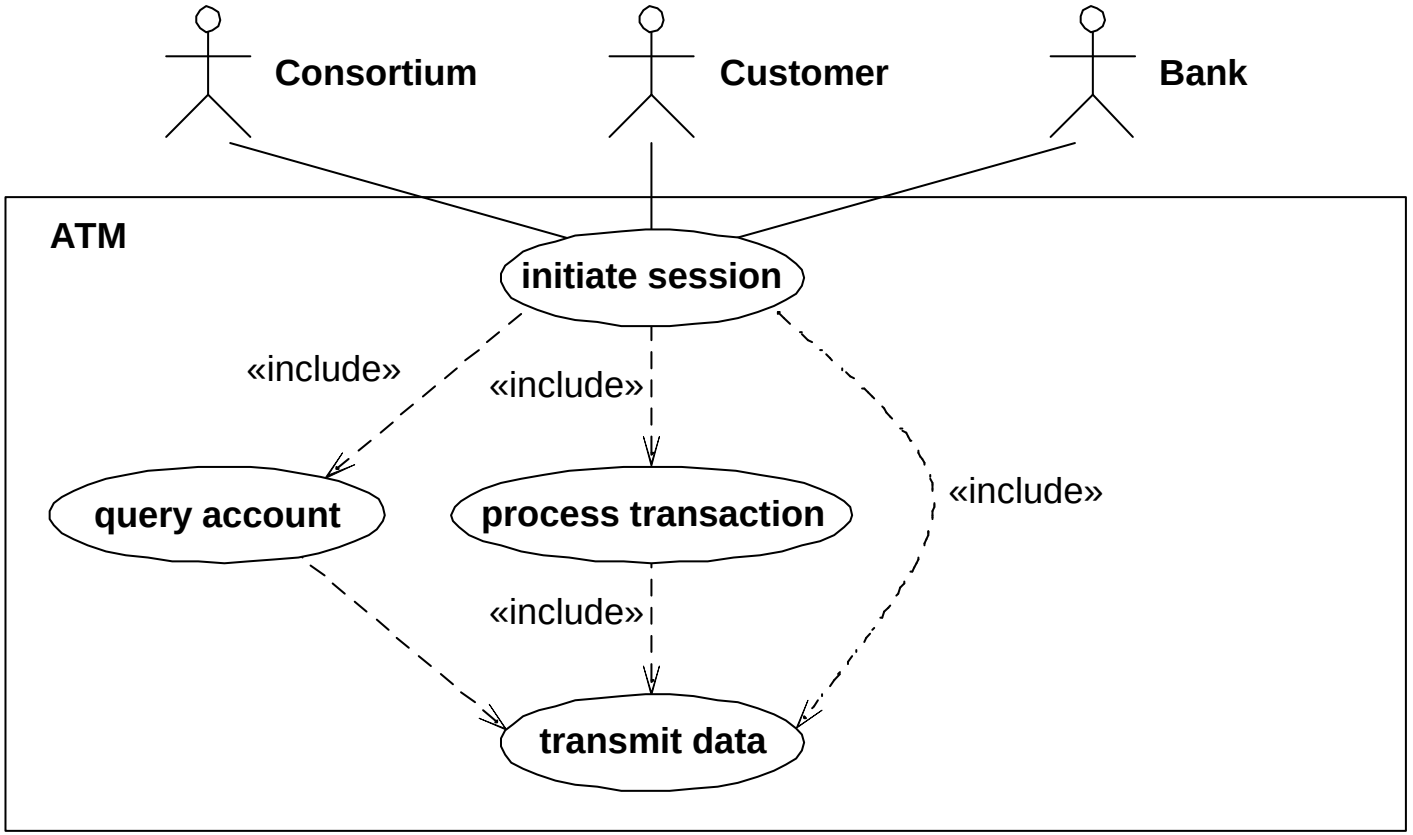
Prepare Activity Diagrams for Use Cases

- ✓ Activity diagrams allow a consolidation of behavior.
- ✓ Sequence diagrams capture interplay between actors, but they do not show alternatives.



Identify dependencies among actors and use cases

- ✓ Include, extend and generalisation relationships (defered untill the base use cases are in place)



Application Class Model (Persistent classes)

- ✓ The actors, use cases, scenarios should be consistent with the **domain class model** (accomplished by testing access paths).
- ✓ Application classes define the application itself rather than domain.
- ✓ Application classes define the way users perceive the application. Such classes are more implementation oriented.

Application Class Model is constructed in following steps:

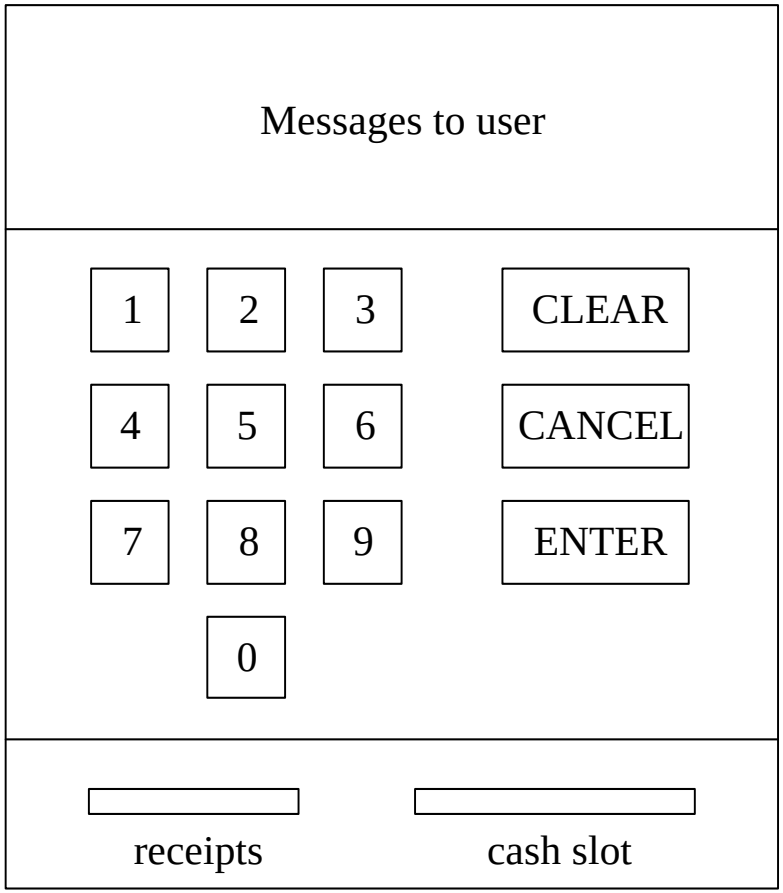
- ✓ Specify User Interfaces.
- ✓ Define Boundary Classes.
- ✓ Determine Control Classes.
- ✓ Check consistency of boundary and control classes with the interaction model.

Specifying User Interfaces

User interface is an object (or group of objects) that provides the user of a system with a coherent way to access its domain objects.

- During analysis emphasis is on **information and control flow**.
- **Requests for a service** are detemined in terms of commands.

User Interface **Layout**



Defining (Interface) Boundary Classes

- ✓ A system must be able to receive information from external sources.
- ✓ Boundary classes isolate a system from the external environment. **Boundary class**:
  - Encapsulates the communication (provides interface boundary) between a system and external actors.
  - Understands the format of messages from actors and converts information for transmission **to** and **from** the system.

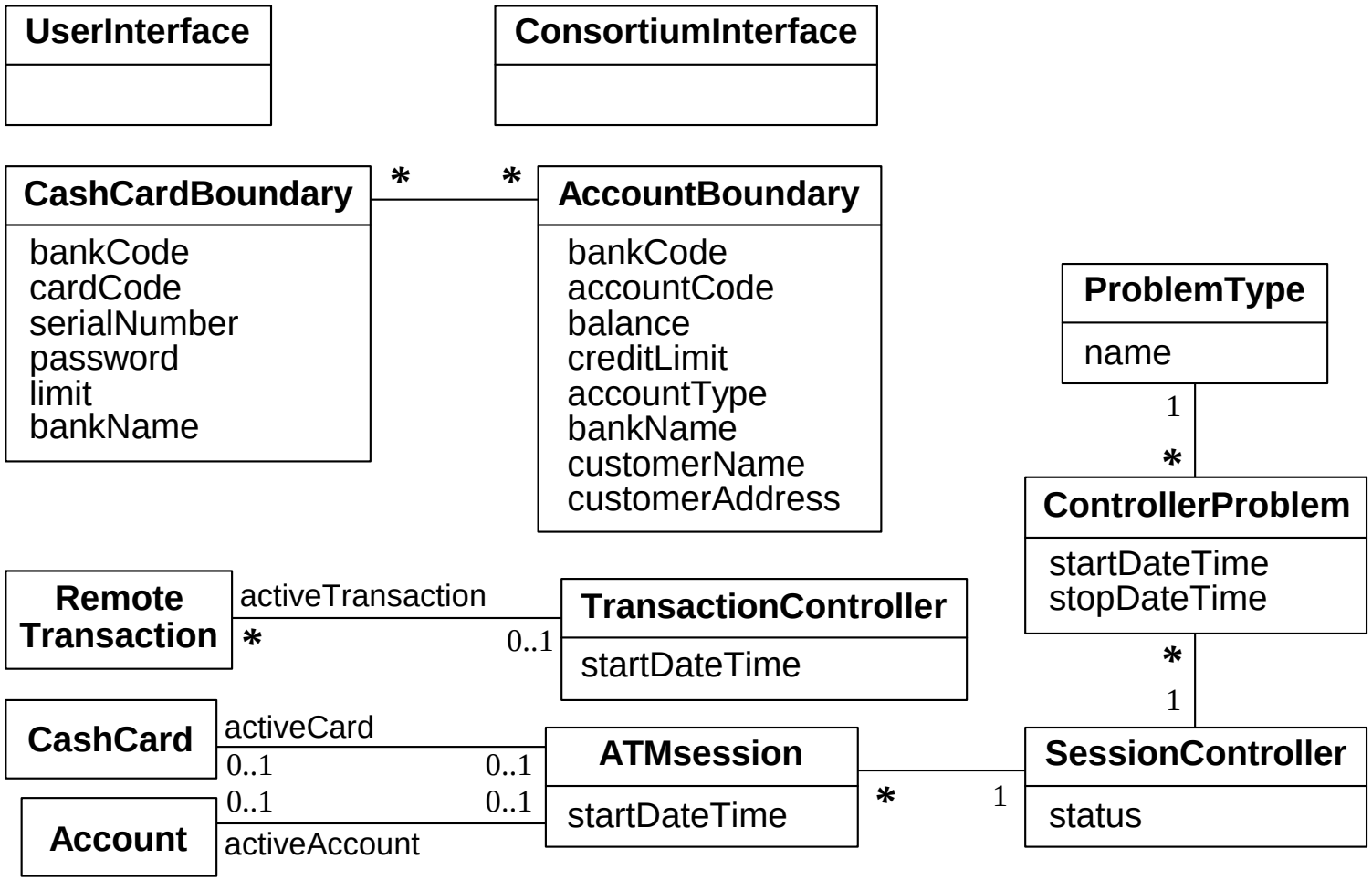
Determining Control Classes

- ✓ A control class object:
  - Manages control within an application.
  - Receives signals from the environment via interface objects and reacts to them by invoking operations on objects in the system.
  - Receives signals from objects within the system and sends signals via interface objects to the environment.
- ✓ A controller object is a reified behavior.
- ✓ Most applications have one or more controllers that sequence the behavior.

Boundary, Control and Domain (Entity, Persistent) classes

- ✓ **Boundary classes**
  - Describe objects that represent the interface between an actor and the system (visual display or sound effect)
  - Often persist beyond a single execution of program
- ✓ **Control classes**
  - Represent transient objects that exist with use case activities
  - Frequently do not persist beyond the program's execution
- ✓ **Entity classes**
  - Describe object that represent the semantics of persistent entities.
  - Often correspond to a data structure in the system database
  - Often persist beyond the program's execution

Examples of Boundary, Control and Domain (Entity, Persistent) classes



Application State Model

- ✓ Determine application classes with states.
- ✓ Identify events.
- ✓ Build state diagrams.
  - Most of work in designing behavior of application classes is modeling their state diagrams.
  - User interface classes and control classes are good candidates for state models (boundary classes tend to be static).
- ✓ Check consistency and completeness with other state diagrams.
- ✓ Check consistency with the class model.
- ✓ Check consistency with the interaction model.

Operations arise from the following sources:

- ✓ Operations from Use Cases
  - Most of functionality is coming from use cases. Use cases lead to activities that can be viewed as sequences of operations at the bottom level of decomposition.
- ✓ Operations from Class Model
  - Access and update (attribute values and association instances) operations have to be shown explicitly.
  - Operations can be meaningful in their own right (e.g.: CreateAccount, CreateCreditCard).

Design: System Architecture determines the **organization** into subsystems

- A **subsystem** is usually identified by the services it provides.
- A **service** is a group of related functions that share some common purpose.
- Subsystem has a well-defined interface with other subsystems.
- **Interface** specifies the form of all interactions and the information flow across subsystem boundaries.
- Each subsystem can be designed independently.
- Most interactions should be internal rather than across subsystem boundaries (**client-server** or **peer-to-peer** relationship between two subsystems).

Making a Reuse Plan

- ✓ Reuse of models is a most practical form of reuse.
- ✓ Only a small fraction of developers should create new models and software components. Most developers reuse them.
- ✓ Models define requirements, therefore they should be carefully organized.
- ✓ Online search of models/components can help, but it is not a substitute for good model organization and maintenance.
- ✓ Reusable Models should be coherent (well-focused themes), complete, consistent, efficient, extensible.

Design

- ✓ OO methodology spans analysis, design and implementation.
- ✓ There is no need to change from one model to another.
- ✓ The design purpose is to bridge the gap from high-level requirements to implementation.
- ✓ Best approach is to carry the analysis classes directly into design.
- ✓ **Class design** is a process of adding more details and keep diagrams consistent with analysis models.

Essence of Design

Requirements, Desired Features

Desired features

The gap

?

Available resources

System infrastructure, Class Library, Components

Realizing Use Cases

- ✓ Elaboration of complex activities (operations), which come from use cases.
- ✓ Use cases describe the required behavior, but they do not define its realization.
- ✓ Design is a process of realizing functionality (bridging the gap).
- ✓ Low level operations are assigned to classes.
- ✓ New design classes should be introduced, if there is no class to hold an operation.

Designing Methods

- ✓ Method is the specification of operation.
- ✓ Methods are defined by using diagrams or pseudocode (which is in fact an algorithm).
- ✓ Method expansions may lead to new classes of objects.
- ✓ The decision of assigning operations to classes is more difficult when more than one object type is involved in an operation.
- ✓ Object oriented operations are associated with object class that is input of operation (recipient of action or manipulated object class rather than the initiator of operation).

Operation Integrity Rules

- ✓ are constraints on the conditions that must always hold before and after operation is executed (part of the method specification).
  - Precondition
    - Mary a couple **If and Only If** this male and female are not married.
  - Postcondition
    - Marriage operation is correctly completed **If and Only If** the male and female are married to each other.

Refactoring

- ✓ It is impossible to get design correct in one pass (inconsistencies, redundancies, etc).
- ✓ Operations or classes may not fully fit all goals.
- ✓ Design may be revisited and reworked so that all operations are conceptually sound.
- ✓ Refactoring is improving design without altering its functionality.
- ✓ In good engineering process is not enough to deliver functionality. To maintain a design, it must be clean, modular and understandable.

Design Optimization

- ✓ Adding Redundant Associations for efficient access.
- ✓ Rearanging execution order.
- ✓ Saving Derived Values to avoid recomputation
  - Derived value (cache) must be updated, if the object on which it depends are changed.
  - Updates are handled in three different ways: explicit update (designer inserts code into update operation of source attributes), periodic recomputation (updates values in bunches), active values (mechanism provided by some languages, updates whenever there is a change).

Reification

- ✓ Reification adds complexity, but you can dramatically expand the flexibility of a system.
- ✓ Reification is the promotion of something that is not an object into object.
- ✓ Behavior can be encoded into an object. Behavior can be transformed and passed to other operations.
- ✓ The entire behavior is encoded in a different language, however, it must be interpreted.
- ✓ Interpretation is much slower than direct execution of code.
  - E.g.: a **Transaction** object is a reified process. It can specialized into **Withdrawing**, **Depositing** and **Transferring**.

Rearraging Classes and Operations

- ✓ Often operations in different classes are similar, but not identical.
- ✓ By adjusting the methods, it is possible to cover them with a single inherited operation.
- ✓ Adjustments:
  - Aligning signatures by adding optional arguments that can be ignored.
  - Implement the special case of operation by calling the general operation (polymorphism).
  - Synonymic attributes in different classes may have different names. Unify the attributes names and move them to the more general class. Operations that access these attributes will match better.

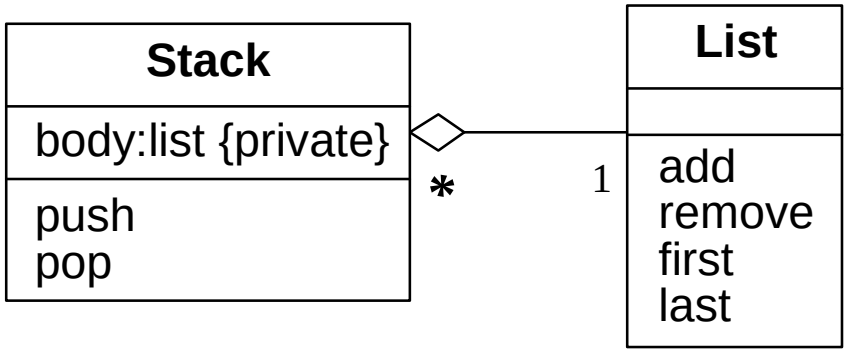
Abstracting out Common Behavior

- When where is **common behavior** in different classes:
  - it is reasonable to create a common superclass for shared features,
  - leaving the specialized features in subclasses.
- It is worthwhile to abstract out a superclass even for one subclass. The superclass may be **reusable** for the future projects.
- Abstract superclasses have benefits beyond sharing and reuse. Specializing a class into few subclasses that separate more general and more specific features is a form of desired **modularity**.

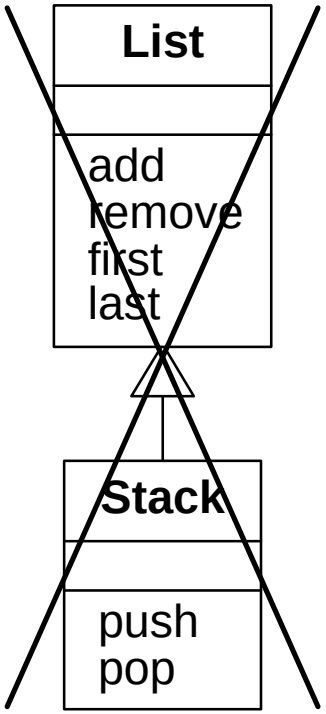
Using Delegation

- ✓ Inheritance of implementation is discouraged
  - Sometimes, the designer is tempted of using inheritance to provide part of behavior for a new class. This lead to problems, if some other operations provide irrelevant behavior.
- ✓ The same goal can be achieved in a safer way by using delegation.
- ✓ The aggregate (composite) object must catch operations and delegate to appropriate part.
- ✓ C++ and Java permit a subclass to inherit a part of superclass operation set and selectively export operations to clients (equivalent to delegation).

Delegation instead of Inheritance as Implementation Technique



Recommended design (with delegation)



Discouraged design (with implementation inheritance)

Information Hiding

- ✓ Methods that know to much about a model are fragile and easily invalidated by changes.
- ✓ Design can be improved by separating external specification and internal methods:
  - An object can access only objects that are directly related by association.
  - An object can access indirectly related objects via the methods of intermediate objects.
  - Minimize class couplings.
  - Use boundary objects to isolate the interior of a system from external environment.
  - Avoid applying a method to the result of another method. Consider a new method that combines two operations.

Coherence Principle

- ✓ A class, operation or package is coherent, if all its parts fit together for achievement of a common goal.
- ✓ A single method should not contain both policy decisions and execution. A method of operation should do one thing well.
- ✓ Separating policy and implementation method increases the possibility of reuse.
- ✓ A class should not serve too many purposes.
- ✓ Classes can be decomposed or specialized, small pieces are more likely to be reusable.
- ✓ Strongly coupled classes should be within a single package. A **coupling strength** can be measured by the number of operations that traverse a given association.