COMP 3111
SOFTWARE ENGINEERING

LECTURE 3
MODELING SOFTWARE SYSTEMS
USING UML
EXERCISE

EXERCISE: BANKING APPLICATION

The following concepts all represent classes, which are related according to the problem statement.

Account	Bank	Checking	CreditCard	Customer
Deposit	Savings	Transaction	Transfer	Withdrawal

Banks issue credit cards (e.g., Visa, MasterCard, etc.) to customers. Each credit card is issued by only one bank to only one customer. Customers hold accounts with banks. Each account is with only one bank and held by only one customer. An account may be a savings account or a checking account. Each savings account has a passbook, which shows the transactions made against the account. Checking accounts do not have passbooks. Customers make transactions against their accounts. A transaction can be a deposit, withdrawal or transfer.

Construct a class diagram, using *only* the ten given classes, that shows, as necessary, associations, aggregations, compositions, generalizations among the classes as well as any necessary association classes and constraints.

EXERCISE: BANKING APPLICATION—SOLUTION

Analysis of Problem Statement

Since we only need to show how the classes are related, the analysis will focus only on discovering the associations, aggregations, compositions and generalizations among the classes and their related multiplicity.

Banks issue credit cards (e.g., Visa, MasterCard, etc.) to customers.

association: Bank *Issues* CreditCard

CreditCard *IsFor* Customer

Each credit card is issued by only one bank to only one customer.

multiplicity: min-card(CreditCard, *Issues*) = 1

max-card(CreditCard, *Issues*) = 1

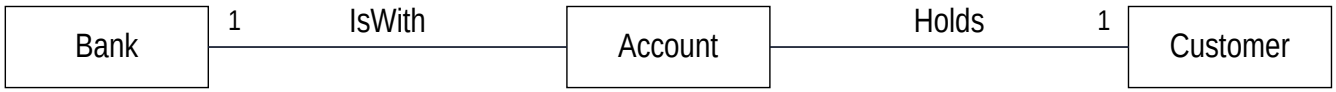
min-card(CreditCard, *IsFor*) = 1

max-card(CreditCard, *IsFor*) = 1

EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

Customers hold accounts with banks.

association: Customer *Holds* Account
Account *IsWith* Bank

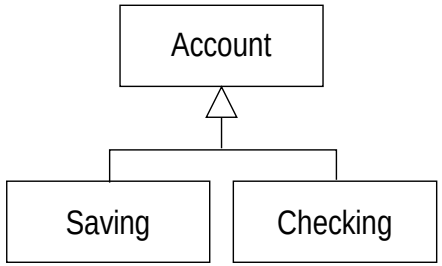


Each account is with only one bank and held by only one customer.

multiplicity: min-card(Account, *IsWith*) = 1
max-card(Account, *IsWith*) = 1
min-card(Account, *Holds*) = 1
max-card(Account, *Holds*) = 1

An account may be a savings account or a checking account.

generalization: Account *generalizes* Saving, Checking



EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

Each savings account has a passbook, which shows the transactions made against the account.

A passbook is a physical object. It should not be represented in the class diagram as the information shown in a passbook can be created from the transactions recorded for a savings account. Moreover, it is not listed as one of the classes that needs to be included in the class diagram.

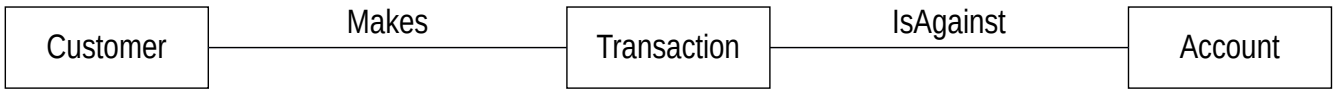
Checking accounts do not have passbooks.

Information about checking accounts; nothing to represent.

Customers make transactions against their accounts.

association: Customer *Makes* Transaction

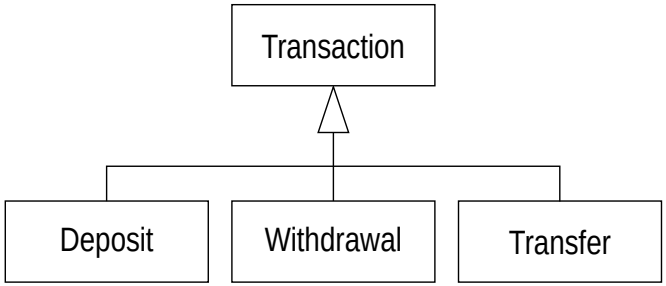
Transaction *IsAgainst* Account



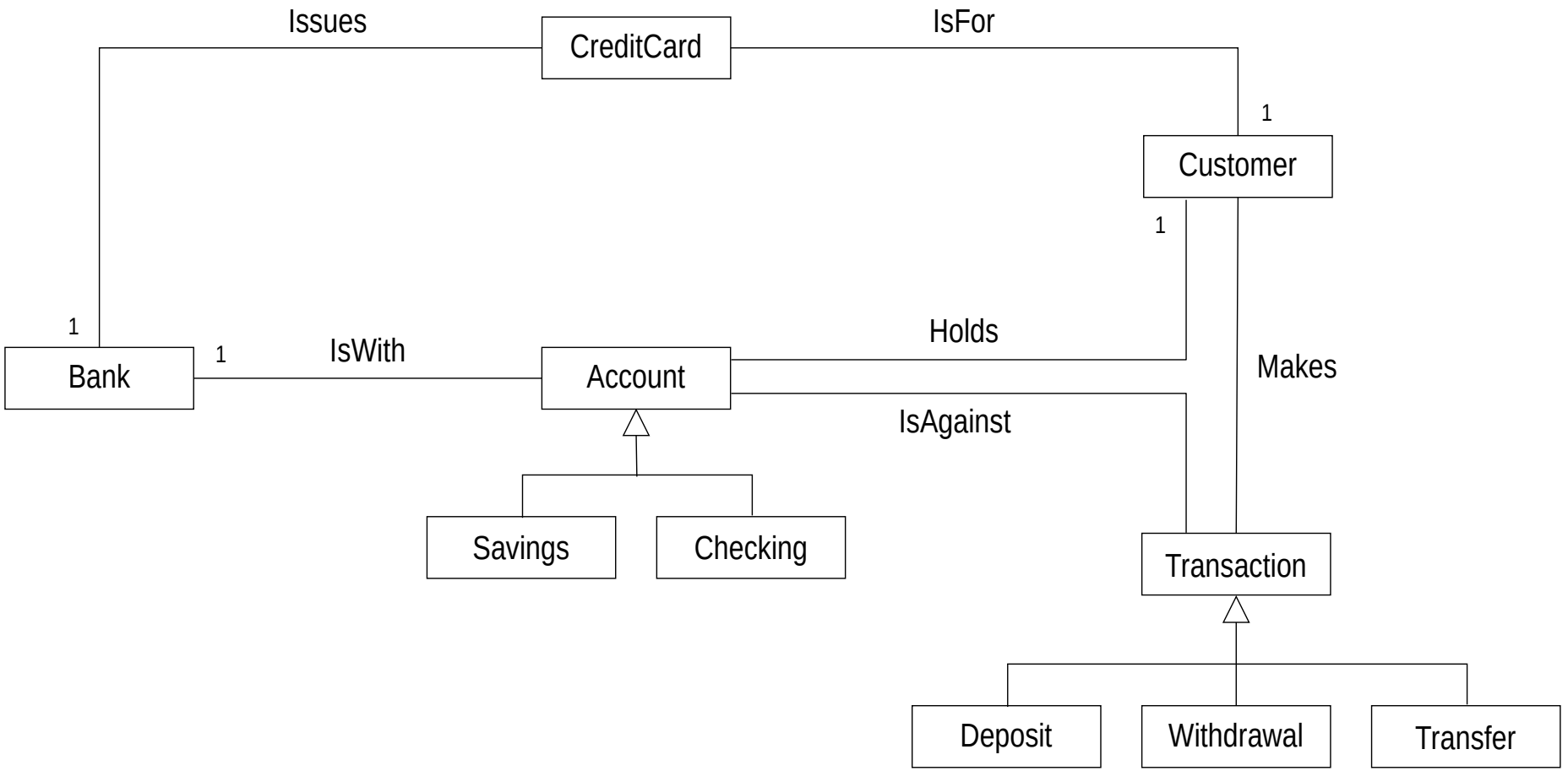
EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

A transaction can be a deposit, withdrawal or transfer.

generalization: Transaction *generalizes* Deposit, Withdrawal, Transfer



EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)



EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

Real World Knowledge

Multiplicity that can be reasonably inferred from real world knowledge.

A bank can issue zero or many credit cards.

multiplicity: min-card(Bank, *Issues*) = 0 max-card(Bank, *Issues*) = *

A customer can have zero or many credit cards.

multiplicity: min-card(Customer, *IsFor*) = 0
max-card(Customer, *IsFor*) = *

A customer may hold no account (they may be a credit card holder only) and can hold many.

multiplicity: min-card(Customer, *Holds*) = 0
max-card(Customer, *Holds*) = *

A bank can have many accounts; whether it needs to have at least one is not known.

multiplicity: min-card(Bank, *IsWith*) = 0 or 1 max-card(Bank, *IsWith*) = *

EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

A customer can make zero or many transactions.

multiplicity: min-card(Customer, *Makes*) = 0
max-card(Customer, *Makes*) = *

Every transaction is made by exactly one customer.

multiplicity: min-card(Transaction, *Makes*) = 1
max-card(Transaction, *Makes*) = 1

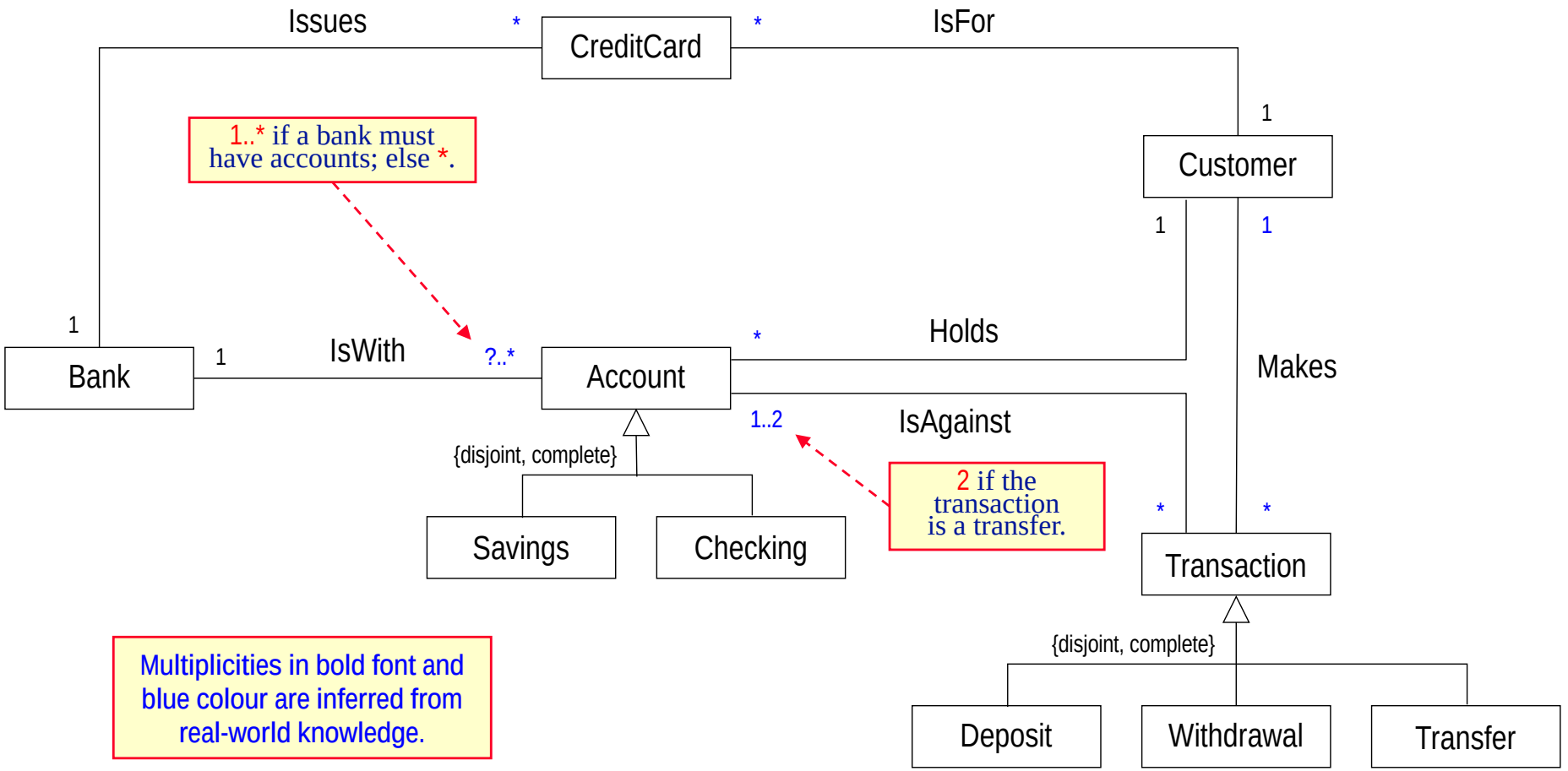
A transaction is against at least one and at most two accounts (if it is a transfer).

multiplicity: min-card(Transaction, *IsAgainst*) = 1
max-card(Transaction, *IsAgainst*) = 2

An account can have zero or many transactions made against it.

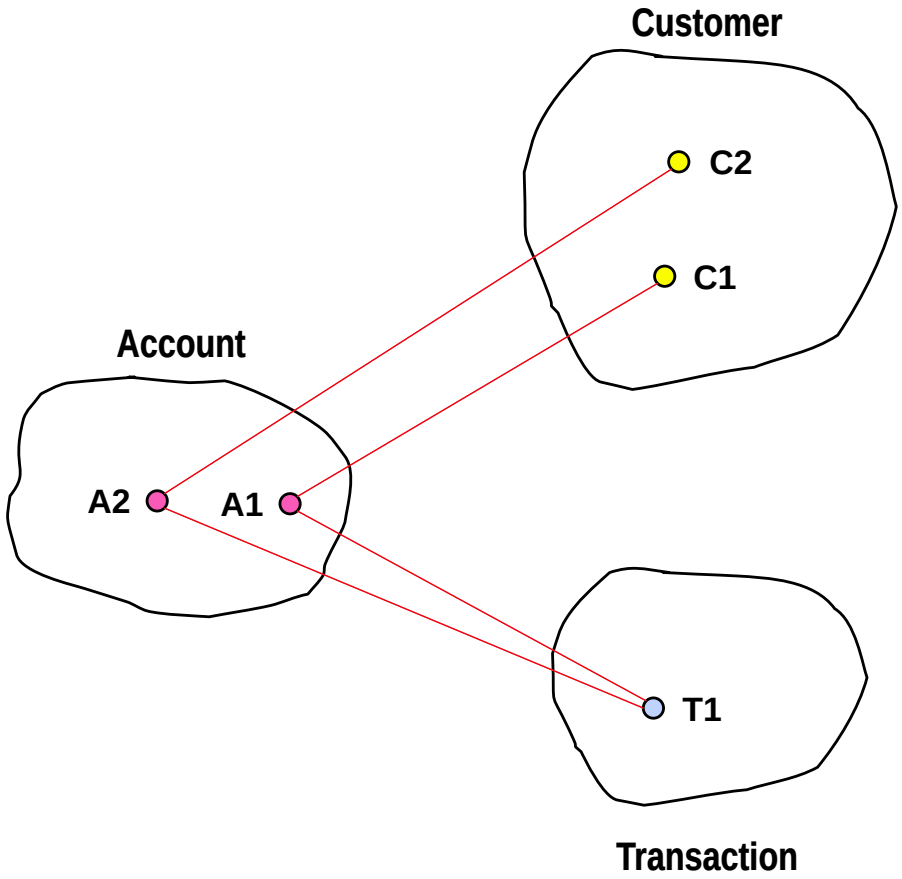
multiplicity: min-card(Account, *IsAgainst*) = 0
max-card(Account, *IsAgainst*) = *

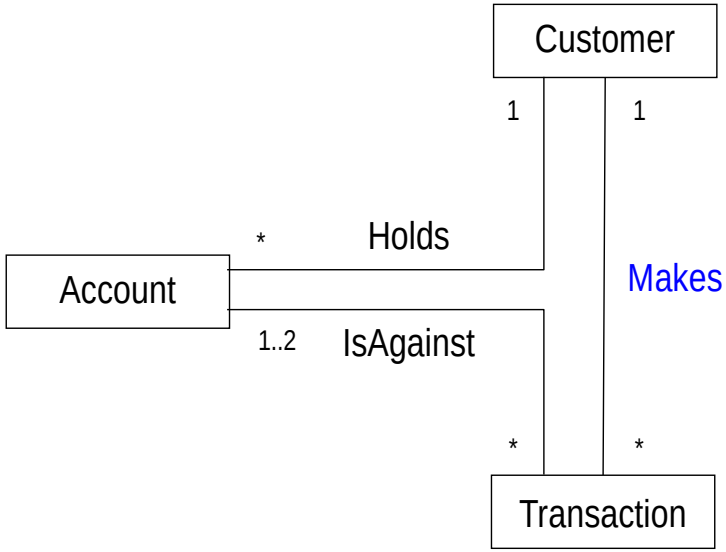
EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)



EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

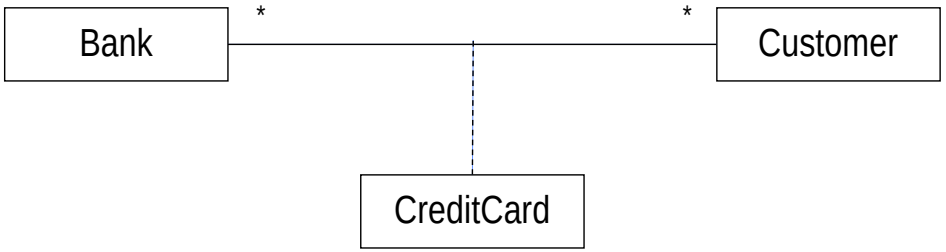
Is the Makes association needed?

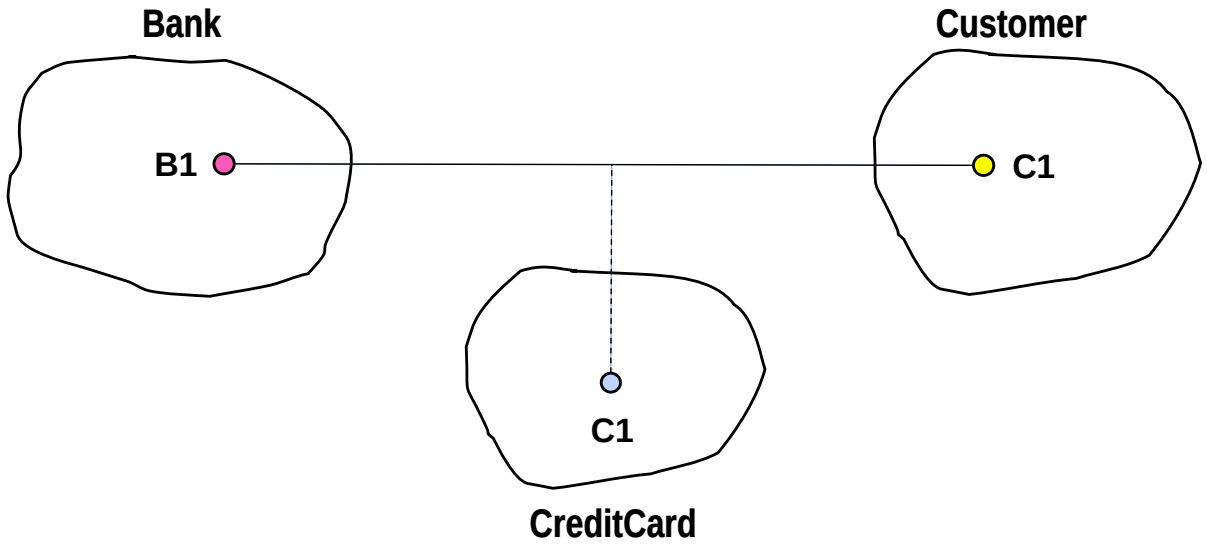




EXERCISE: BANKING APPLICATION—SOLUTION (cont'd)

Is this representation OK?





EXERCISE: BANKING APPLICATION — COMMON ERRORS

- Missing/incorrect classes/associations
- Missing association names
  - While associations names are not required, it is always a good idea to name them.
- Incorrect use of aggregation
  - There is no aggregation in this example!
- Incorrect use of generalization
  - Ask “kind of” question.
- Including operations as associations
  - Associations do not represent operations, but result of operations.
- Using modeling elements incorrectly
  - (e.g., xor)