FPT Education

FPT UNIVERSITY

SOFTWARE DESIGN (SWD392)

CH14 - DESIGNING OBJECT-ORIENTED SOFTWARE ARCHITECTURES

Main Contents

- General Introduction
- Concepts, Architectures, and Patterns
- Designing Information Hiding Classes
- Designing Class Interface & Operations
- Data Abstraction Classes
- State-Machine Classes
- Graphical User Interaction Classes
- Business Logic Classes
- Inheritance in Design
- Class Interface Specifications
- Detailed Design of Information Hiding Classes
- Polymorphism and Dynamic Binding

General Introduction

- Object-oriented design refers to software systems that are designed using the concepts of
  - ***Information hiding***: encapsulate different kinds of information (details of a data structure, state machine,..)
  - ***Classes***: the design of class interfaces and the operations provided by each class
  - ***Inheritance***: mechanism for sharing and reusing code between classes - a child class inherits the properties (encapsulated data and operations) of a parent class.
- Objects are instantiated from classes & are accessed through operations
  - Are also referred to as methods
  - The specification and the implementation of a function performed by an object

Concepts, Architectures, and Patterns

- ***Information hiding*** is a fundamental design concept in which a class encapsulates some information, such as a data structure, that is hidden from the rest of the system
- The separation of the interface from the implementation.
- The interface forms a contract between the provider of the interface and the user of the interface
- These object-oriented concepts have also been applied and extended in the design of
  - Distributed and component-based software architectures,
  - Concurrent and real-time software architectures,
  - Service-oriented architectures,
  - Software product line architectures.
- For communication between objects, the Call/Return pattern is the only pattern of communication in a sequential architecture

Designing Information Hiding Classes

- In design modeling, information hiding classes are categorized by stereotype.
- Classes determined from the analysis model are categorized as following:
  - Entity classes from the analysis model, encapsulate data.
  - Boundary classes Communicate with and interface to the external environment.
    - Active (concurrent) classes: device I/O classes, proxy classes
    - Passive boundary classes: graphical user interaction class
  - Control classes. Provide the overall coordination for a collection of objects.
    - Control classes are often active (concurrent) classes
    - One of the passive control classes is the state-machine class, which encapsulates a finite state machine.
  - Application logic classes.
    - Encapsulate application-specific logic and algorithms
    - Categorized as business logic classes, service classes, or algorithm classes.

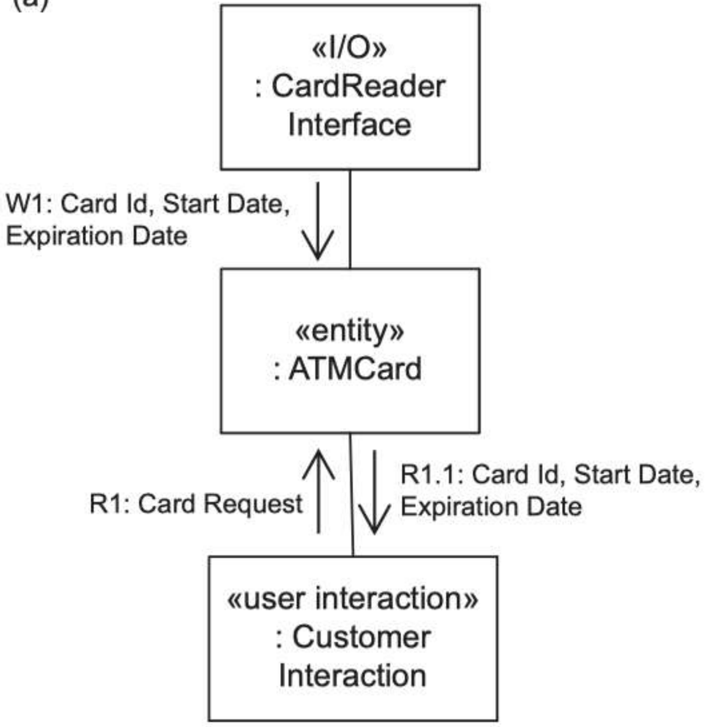
Designing Class Interface & Operations

- The class interface consists of the operations (methods) provided by each class.
- Each operation can have input parameters, output parameters, and (if it is a function) a return value.
- The operations of a class can be determined typically from the dynamic model
    - Interaction message: operations being invoked at the destination object receiving the message
    - Message passing between passive objects consists of an operation in one object invoking an operation provided by another object
- From the class diagrams of the static model: standard operations are create, read, update, delete (CRUD actions)

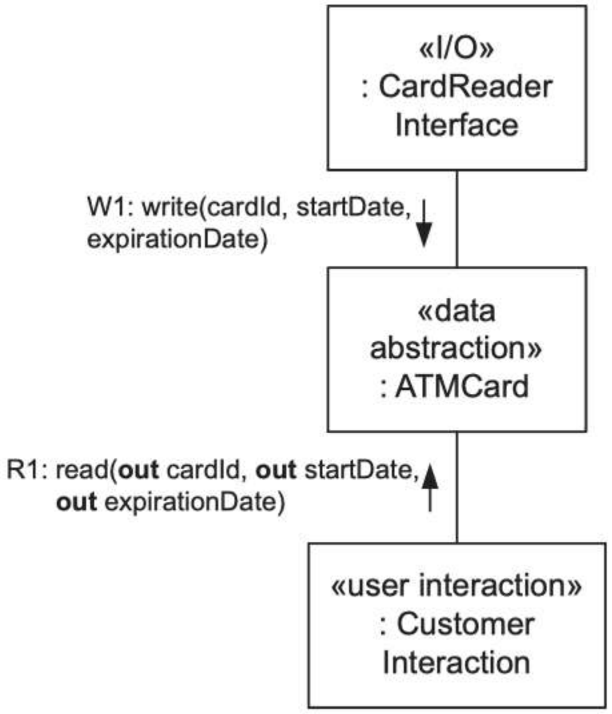
Designing Class Interface & Operations

Designing Class Operations from the Interaction Model

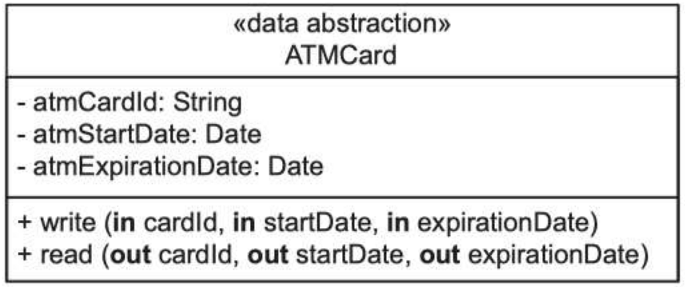
(a)



(b)



(c)


«data abstraction»
ATMCard
- atmCardId: String
- atmStartDate: Date
- atmExpirationDate: Date
+ write (**in** cardId, **in** startDate, **in** expirationDate)
+ read (**out** cardId, **out** startDate, **out** expirationDate)

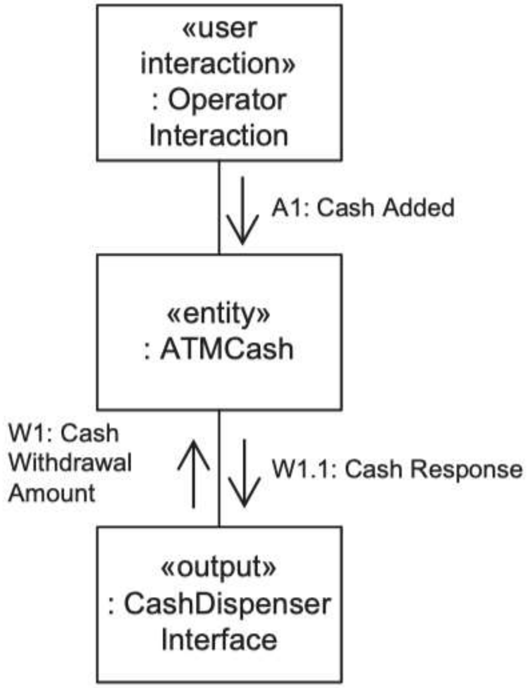
(a) Analysis model: communication diagram.
(b) Design model: communication diagram.
(c) Design model: class diagram

Data Abstraction Classes

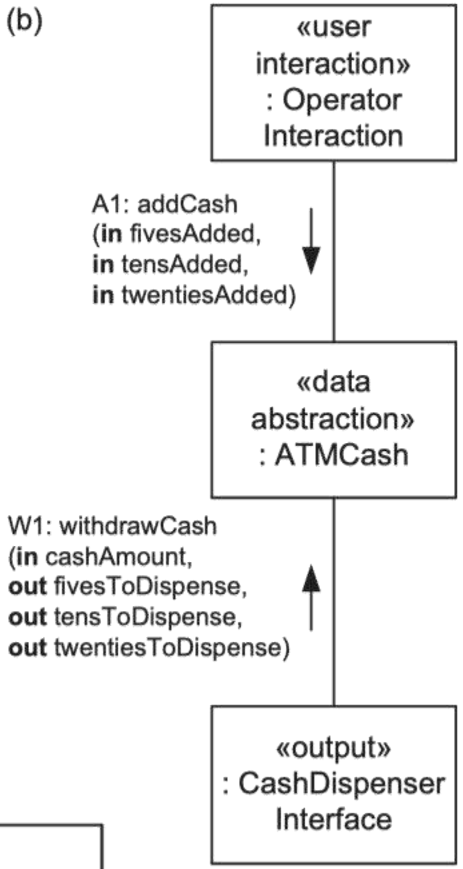
- Used to encapsulate the data structure
  - Hiding the internal details of how the data structure is represented
  - The operations are designed as access procedures or functions whose internals, which define how the data structure is manipulated, are also hidden.
- Each entity class in the analysis model that encapsulates data is designed as a data abstraction class.
- An entity class stores some data and provides operations to access the data and to read or write to the data.
- In the data abstraction class
  - The information on the attributes information should be available from the static model of the problem domain
  - The operations of the data abstraction class are determined by considering the needs of the client objects that use the data abstraction object in order to indirectly access the data structure. This can be determined by analyzing how the data abstraction object is accessed by other objects, as given in the communication model.

Data Abstraction Classes

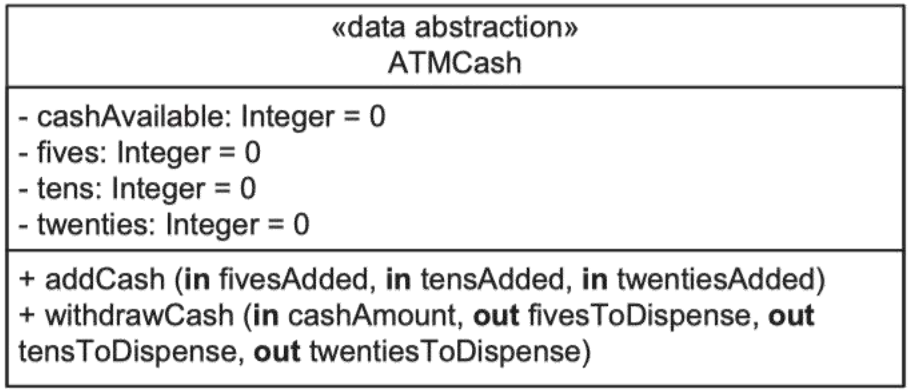
(a)


«user interaction» : Operator Interaction
A1: Cash Added
«entity» : ATMCash
W1: Cash Withdrawal Amount
W1.1: Cash Response
«output» : CashDispenser Interface

(b)

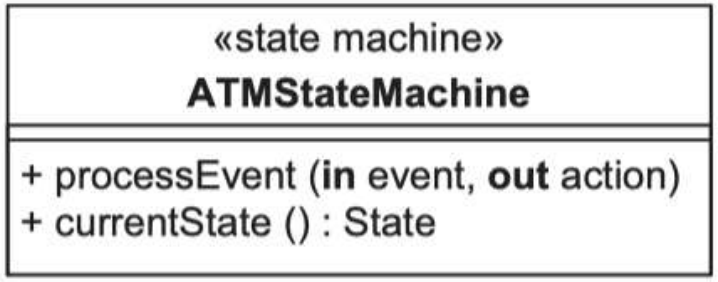

«user interaction» : Operator Interaction
A1: addCash (**in** fivesAdded, **in** tensAdded, **in** twentiesAdded)
«data abstraction» : ATMCash
W1: withdrawCash (**in** cashAmount, **out** fivesToDispense, **out** tensToDispense, **out** twentiesToDispense)
«output» : CashDispenser Interface

(c)


«data abstraction» ATMCash
- cashAvailable: Integer = 0
- fives: Integer = 0
- tens: Integer = 0
- twenties: Integer = 0
+ addCash (**in** fivesAdded, **in** tensAdded, **in** twentiesAdded)
+ withdrawCash (**in** cashAmount, **out** fivesToDispense, **out** tensToDispense, **out** twentiesToDispense)

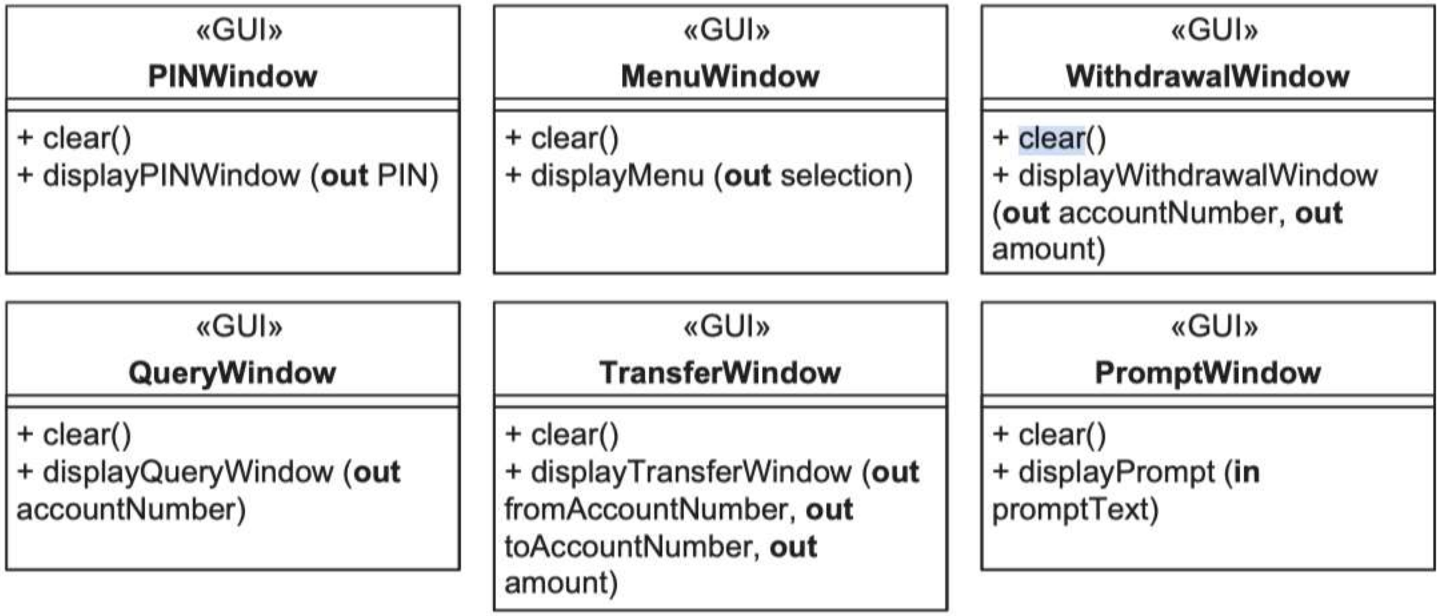
State-Machine Classes

- During class design, the state-machine class determined in the analysis model is designed.
- The statechart executed by the state-machine object is encapsulated in a state transition table. Thus, the state-machine class hides the contents of the state transition table and maintains the current state of the object
- State-machine classes provide the operations that access the state transition table & change the state of the object.


«state machine»
ATMStateMachine
+ processEvent (in event, out action)
+ currentState () : State

Graphical User Interaction Classes

- A graphical user interaction (GUI) class hides from other classes the details of the interface to the user.
- In a given application, the user interface might be a simple command line interface or a sophisticated GUI
- A command line interface is typically handled by one user interaction class.

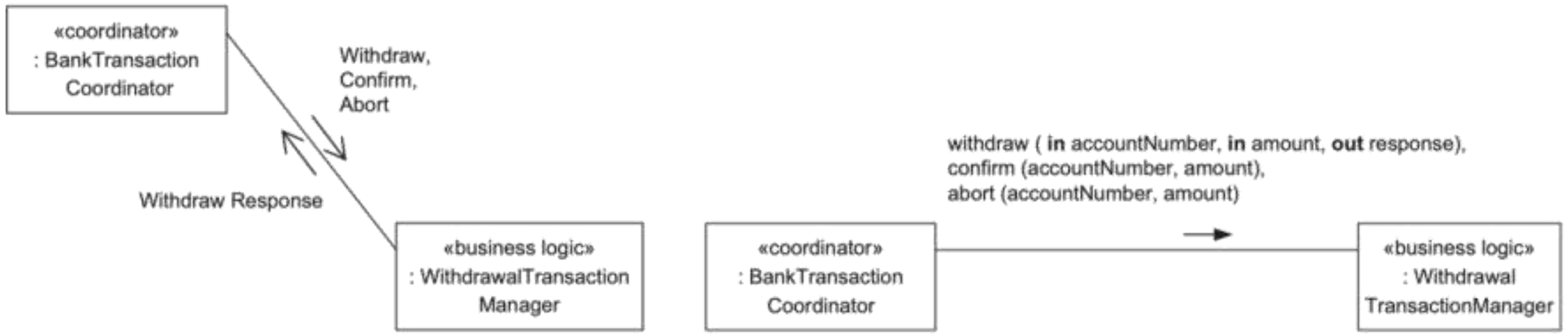


«GUI» PINWindow
+ clear()
+ displayPINWindow (out PIN)
«GUI» MenuWindow
+ clear()
+ displayMenu (out selection)
«GUI» WithdrawalWindow
+ clear()
+ displayWithdrawalWindow (out accountNumber, out amount)
«GUI» QueryWindow
+ clear()
+ displayQueryWindow (out accountNumber)
«GUI» TransferWindow
+ clear()
+ displayTransferWindow (out fromAccountNumber, out toAccountNumber, out amount)
«GUI» PromptWindow
+ clear()
+ displayPrompt (in promptText)

Business Logic Classes

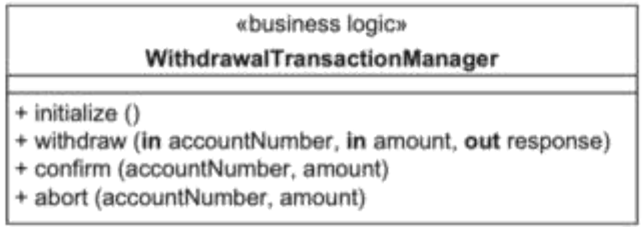
A **business logic class** defines the decision-making, business-specific application logic for processing a client's request. The goal is to encapsulate business rules that could change independently of each other into separate business logic classes.

Usually a business logic object accesses various entity objects during its execution.



«coordinator» : BankTransaction Coordinator
Withdraw, Confirm, Abort
Withdraw Response
«business logic» : WithdrawalTransaction Manager
«coordinator» : BankTransaction Coordinator
withdraw (in accountNumber, in amount, out response), confirm (accountNumber, amount), abort (accountNumber, amount)
«business logic» : Withdrawal TransactionManager

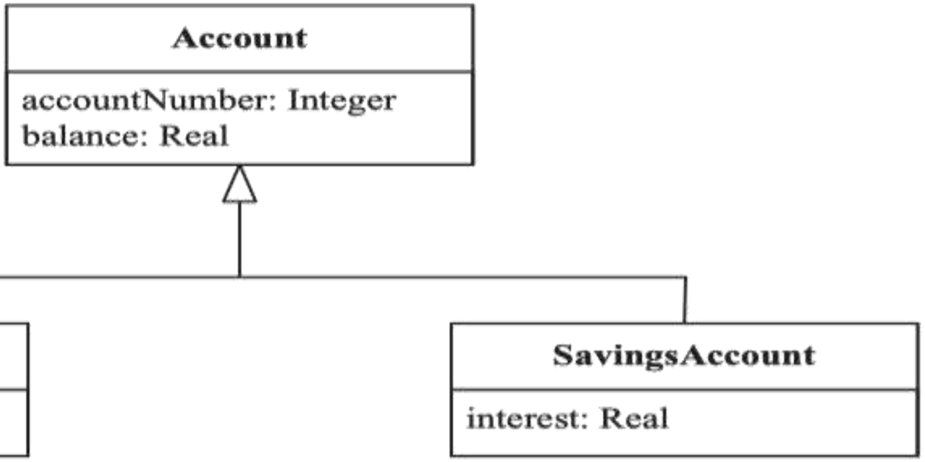
(c)



«business logic»
WithdrawalTransactionManager
+ initialize ()
+ withdraw (**in** accountNumber, **in** amount, **out** response)
+ confirm (accountNumber, amount)
+ abort (accountNumber, amount)

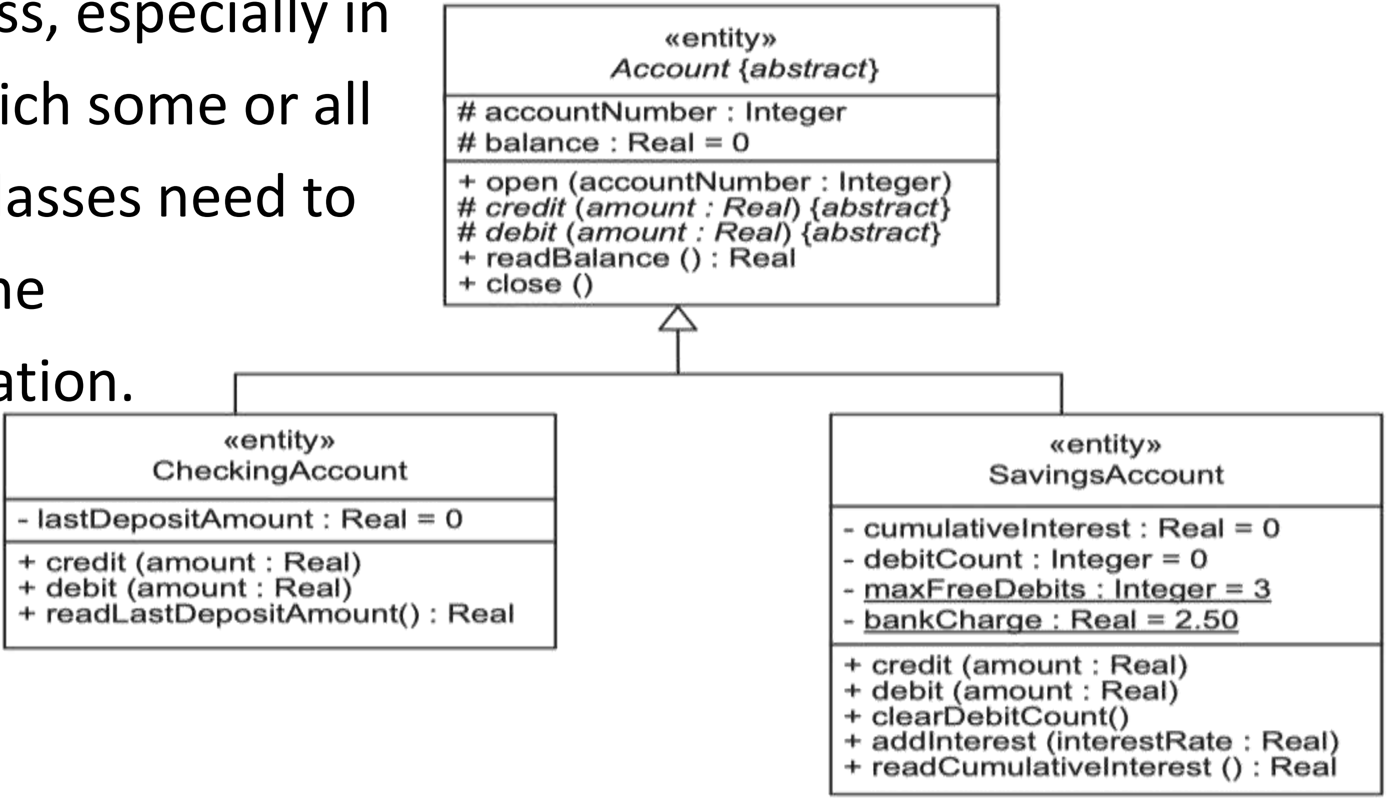
Inheritance in Design

- Inheritance can be used when designing two similar, but not identical, classes (share many, but not all, characteristics).
- Inheritance can also be used when adapting a design for either maintenance or reuse purposes.
- Class Hierarchies
  - Class hierarchies (also referred to as generalization/specialization hierarchies and inheritance hierarchies) can be developed either top-down, bottom-up, or by some combination of the two approaches.
  - When designing with inheritance, the internals of the parent classes are visible to the subclasses



Inheritance in Design
Abstract Classes

- An abstract class is a class with no instances, used as a template to create subclasses
- An abstract operation is an operation that is declared in an abstract class but not implemented.
- In reality, some of the operations may be implemented in the abstract class, especially in cases in which some or all of the subclasses need to use the same implementation.


«entity»
Account {abstract}
# accountNumber : Integer
# balance : Real = 0
+ open (accountNumber : Integer)
# credit (amount : Real) {abstract}
# debit (amount : Real) {abstract}
+ readBalance () : Real
+ close ()
«entity»
CheckingAccount
- lastDepositAmount : Real = 0
+ credit (amount : Real)
+ debit (amount : Real)
+ readLastDepositAmount() : Real
«entity»
SavingsAccount
- cumulativeInterest : Real = 0
- debitCount : Integer = 0
- maxFreeDebits : Integer = 3
- bankCharge : Real = 2.50
+ credit (amount : Real)
+ debit (amount : Real)
+ clearDebitCount()
+ addInterest (interestRate : Real)
+ readCumulativeInterest () : Real

Class Interface Specifications

A class interface specification defines the interface of the information hiding class, including the specification of the operations provided by the class. It defines the following:

- Information hidden by information hiding class: for example, data structure(s) encapsulated, in the case of a data abstraction class.
- Class structuring criteria used to design this class.
- Assumptions made in specifying the class: for example, whether one operation needs to be called before another.
- Anticipated changes. This is to encourage consideration of design for change.
- Superclass (if applicable)
- Inherited operations (if applicable)
- Operations provided by the class. For each operation, define:
  - Function performed
  - Precondition (a condition that must be true when the operation is invoked)
  - Postcondition (a condition that must be true at the completion of the operation)
  - Invariant (a condition that must be true at all times: before, during, and after execution of the operation)
  - Input parameters
  - Output parameters
  - Operations used from other classes

FPT Education
FPT UNIVERSITY

Class Interface Specifications

Example of Class Interface Specification 1/2

Information Hiding Class: CheckingAccount
Information Hidden: Encapsulates checking account attributes and their current values.
Class structuring criterion: Data abstraction class
Assumptions: Checking accounts do not have interest.
Anticipated changes: Checking accounts may be allowed to earn interest.
Superclass: Account
Inherited operations: open, credit, debit, readBalance, close
Operations provided:

1. credit (**in** amount : Real)

   Function: Adds the amount credited to the current balance and stores it as the amount last deposited.
   Precondition: Account has been created.
   Postcondition: Checking account has been credited.
   Input parameters: amount – funds to be added to account
   Operations used: None

FPT Education
FPT UNIVERSITY

Class Interface Specifications

Example of Class Interface Specification 2/2

2. debit (**in** amount : Real)

   Function: Deducts amount from balance.
   Precondition: Account has been created.
   Postcondition: Checking account has been debited.
   Input parameters: amount – funds to be deducted from account
   Output parameters: None
   Operations used: None

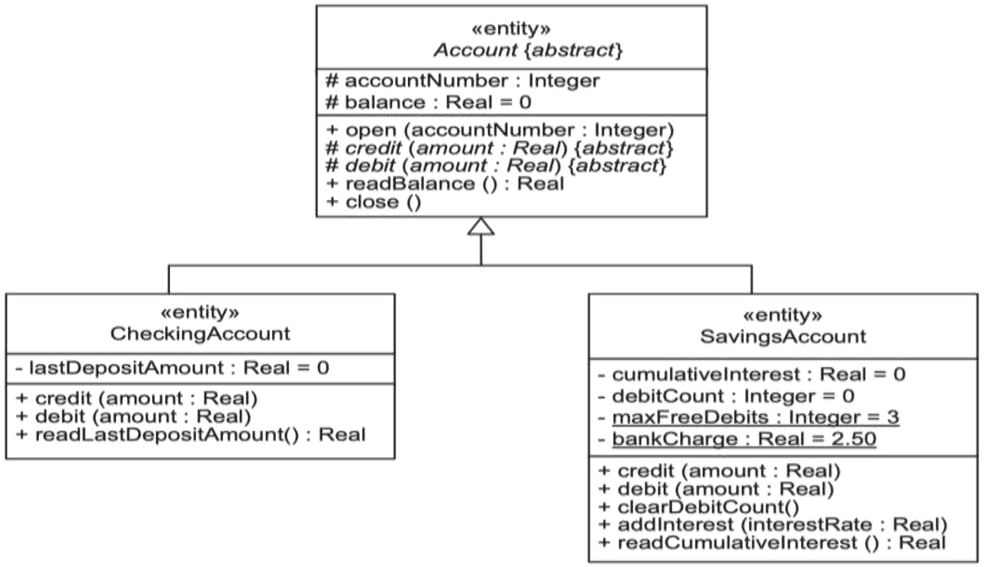
3. readLastDepositAmount (): Real

   Function: Returns the amount last deposited into the account.
   Precondition: Account exists.
   Invariant: Values of account attributes remain unchanged.
   Postcondition: Amount last deposited into the account has been read.
   Input parameter: None
   Output parameters: Amount last deposited into the account
   Operations used: None

Detailed Design of Information Hiding Classes

During **detailed design of the information hiding classes**, the internal algorithmic design of each operation is determined. The operation internals are documented in *pseudocode*.

Detailed Design of the Account Abstract Superclass

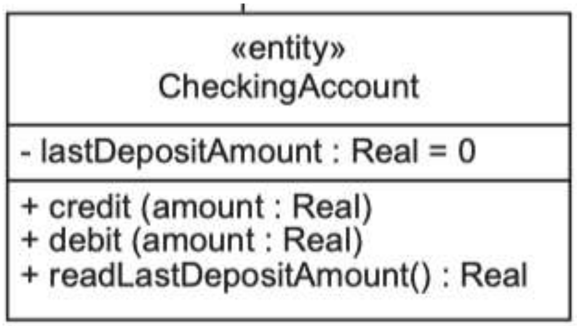


«entity»
Account {abstract}
# accountNumber : Integer
# balance : Real = 0
+ open (accountNumber : Integer)
# *credit (amount : Real) {abstract}*
# *debit (amount : Real) {abstract}*
+ readBalance () : Real
+ close ()
«entity»
CheckingAccount
- lastDepositAmount : Real = 0
+ credit (amount : Real)
+ debit (amount : Real)
+ readLastDepositAmount() : Real
«entity»
SavingsAccount
- cumulativeInterest : Real = 0
- debitCount : Integer = 0
- maxFreeDebits : Integer = 3
- bankCharge : Real = 2.50
+ credit (amount : Real)
+ debit (amount : Real)
+ clearDebitCount()
+ addInterest (interestRate : Real)
+ readCumulativeInterest () : Real

```
■ Attributes:
     accountNumber, balance
■ Operations:
  • open (in accountNumber : Integer)
       begin;
            create new account;
            assign accountNumber;
            set balance to zero;
       end.
  • close ()
       begin; close the account; end.
■ readBalance () : Real
     begin; return value of balance; end.
■ credit (in amount : Real)
     Defer implementation to subclass.
■ debit (in amount : Real)
     Defer implementation to subclass.
```

Detailed Design of Information Hiding Classes

Detailed Design of Checking Account Subclass



- Attributes:
  - Inherit: accountNumber, balance
  - Declare: lastDepositAmount
- Operations:
  - Inherit specification and implementation: open, close, readBalance
  - Inherit specification and define implementation:

```
credit(in amount : Real);
    begin;
        Add amount to balance;
        Set lastDepositAmount equal to amount;
    end.
```

  - Inherit specification and define implementation of:

```
debit (in amount : Real);
    begin;
        Deduct amount from balance;
    end.
```

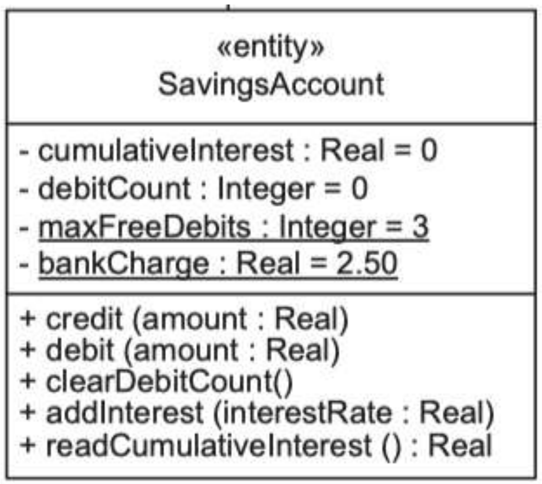
  - Add operation:

```
readLastDepositAmount () : Real
    begin;
        return value of lastDepositAmount;
    end.
```

FPT Education
FPT UNIVERSITY

Detailed Design of Information Hiding Classes

Detailed Design of Savings Account Subclass 1/2


«entity»
SavingsAccount
- cumulativeInterest : Real = 0
- debitCount : Integer = 0
- maxFreeDebits : Integer = 3
- bankCharge : Real = 2.50
+ credit (amount : Real)
+ debit (amount : Real)
+ clearDebitCount()
+ addInterest (interestRate : Real)
+ readCumulativeInterest () : Real

- Attributes:
  - Inherit: accountNumber, balance
  - Declare: cumulativeInterest, debitCount
  - Declare static class attributes: maxFreeDebits, bankCharge
- Operations:
  - Inherit specification and implementation: open, close, and readBalance.
  - Inherit specification and redefine implementation:

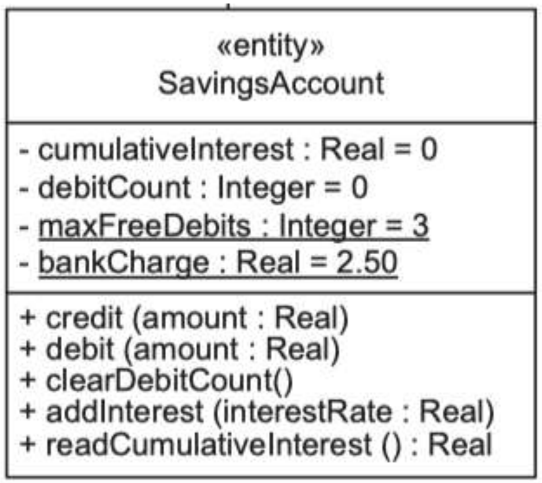
```
debit (in amount : Real);
    begin
        Deduct amount from balance;
        Increment debitCount;
        if maxFreeDebits > debitCount
            then deduct bankCharge from balance;
    end.
```

  - Inherit specification and redefine implementation:

```
credit(in amount : Real);
   begin; add amount to balance; end.
```

Detailed Design of Information Hiding Classes

Detailed Design of Savings Account Subclass 2/2


«entity»
SavingsAccount
- cumulativeInterest : Real = 0
- debitCount : Integer = 0
- maxFreeDebits : Integer = 3
- bankCharge : Real = 2.50
+ credit (amount : Real)
+ debit (amount : Real)
+ clearDebitCount()
+ addInterest (interestRate : Real)
+ readCumulativeInterest () : Real

- Declared operations:

```
addInterest (interestRate : Real)
    begin
        Compute dailyInterest = balance * interestRate;
        Add dailyInterest to cumulativeInterest and to balance;
    end.
```

- readCumulativeInterest () : Real

```
begin; return value of cumulativeInterest; end.
```

- clearDebitCount (),

```
begin; Reset debitCount to zero; end.
```

Polymorphism and Dynamic Binding

- Polymorphism is Greek for “many forms”. In object-oriented design, polymorphism is used to mean that different classes may have the same operation name. The specification of the operation is identical for each class; however, classes can implement the operation differently.
- Dynamic binding is used in conjunction with polymorphism.
- Dynamic binding means that the association of a request to an object’s operation is done at run-time and can thus change from one invocation to the next.

Polymorphism and Dynamic Binding

Example of Polymorphism and Dynamic Binding

```
begin
private anAccount: Account;
Prompt customer for account type and withdrawal amount
if customer responds checking
    then – assign customer's checking account to anAccount
        . . .
        anAccount := customerCheckingAccount;
        . . .
    elseif customer responds savings
        then – assign customer's savings account to anAccount
        . . .
        anAccount := customerSavingsAccount;
        . . .
    endif;
    . . .
    – debit an Account, which is a checking or savings account
    anAccount.debit (amount);
    . . .
end;
```

Q
A