AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

SOFTWARE ENGINEERING PRINCIPLES
SOFTWARE SPECIFICATION:
CASH DISPENSER

STEFAN HALLESTEDE
PETER GORM LARSEN
CARL SCHULTZ

REQUIREMENTS 1

There are many tills, which can access a central resource containing the detailed records of customers' bank accounts. Customers insert a card into the till and type in a PIN (Personal Identification Number), which is encoded by the till and compared with a code stored on the card.

After successfully identifying themselves to the system, customers may try to:

1. view the balance of their accounts;
2. make a withdrawal of cash;
3. ask for a statement of their account to be sent by post.

REQUIREMENTS 2

Information on accounts is held in a central database and may be unavailable. In this case (1) above may not be possible. If the database is available, any amount up to the total in the account may be withdrawn, subject to a fixed daily limit on withdrawals. This means that the amount withdrawn within the day must be stored on the card. 'Illegal' cards are kept by the till. The solution should consider feasible implementation architectures. These are likely to include a data line between each till and the central database. For example, transactions at a till may be recorded locally and sent in a queue down the line to the central database.

TYPES: ACCOUNT + TRANSACTION

```
types
  Account :: cards : map CardId to Cardholder
             balance : nat
             transactions : seq of Transaction
  inv account ==
    TransactionsInvariant(account.transactions);

  Transaction :: date : Date
                 cardId : CardId
                 amount : nat;
```

TYPES: ALL OTHER TYPES

```
Card :: code : Code
        cardId : CardId
        accountId : AccountId;

Cardholder :: name : Name;
AccountId = nat; -- could also be token
Name = seq of char;
CardId = nat;
Code = nat;
PinCode = nat;
Date = seq of char;
```

ACCOUNT INVARIANT

functions

```
TransactionsInvariant : seq of Transaction +> bool
TransactionsInvariant(ts) ==
  forall date in set {t.date | t in seq ts} &
    DateTotal(date,ts) <= dailyLimit;

DateTotal : Date * seq of Transaction +> nat
DateTotal(date,ts) ==
  Sum([t.amount | t in seq ts & t.date = date]);
```

STATE FOR CASH DISPENSER

```
state System of
  accounts : map AccountId to Account
  illegalCards : set of CardId
  curCard : [Card]
  cardOk : bool
  retainedCards : set of Card
inv mk_System(accs,-,curCard,cardOk,-) ==
      (curCard = nil => not cardOk) and
      (forall id1, id2 in set dom accs &
        id1 <> id2 =>
        dom accs(id1).cards inter
        dom accs(id2).cards = {})
init s == s = mk_System({|->},{},nil,false,{})
end
```

INSERTING A CARD

operations

```
InsertCard : Card ==> ()
InsertCard(c) ==
  curCard := c
pre curCard = nil;
```

VALIDATING A CARD

```
Validate: PinCode ==> <PinOk> | <PinNotOk> | <Retained>
Validate(pin) ==
  let codeOk = curCard.code = Encode(pin),
      cardLegal = IsLegalCard(curCard,illegalCards,
                              accounts)
  in
    (if not cardLegal then
      (retainedCards := retainedCards union {curCard};
       cardOk := false;
       curCard := nil;
       return <Retained>)
     else
       cardOk := codeOk;
     return if cardOk
            then <PinOk>
            else <PinNotOk>)
pre curCard <> nil and not cardOk;
```

RETURN CARD AND GET BALANCE

```
ReturnCard : () ==> ()
ReturnCard() ==
  (cardOk := false;
   curCard:= nil)
pre curCard <> nil;

GetBalance : () ==> nat
GetBalance() ==
  return accounts(curCard.accountId).balance
pre curCard <> nil and cardOk and
    IsLegalCard(curCard,
                illegalCards,
                accounts);
```

MAKING A WITHDRAWAL

```
MakeWithdrawal : nat * Date ==> bool
MakeWithdrawal(amount,date) ==
  let mk_Card(-,cardId,accountId) = curCard,
      transaction = mk_Transaction(date,cardId,amount)
  in
    if accounts(accountId).balance - amount >= 0 and
      DateTotal(date,accounts(accountId).transactions^
                                        [transaction])
      <= dailyLimit
    then
      (accounts(accountId).balance :=
         accounts(accountId).balance - amount;
       accounts(accountId).transactions :=
         accounts(accountId).transactions ^ [transaction];
       return true)
   else
     return false
pre curCard <> nil and cardOk and
    IsLegalCard(curCard,illegalCards,accounts);
```

IS A CARD VALID

functions

```
IsLegalCard : Card * set of CardId *
              map AccountId to Account -> bool
IsLegalCard(mk_Card(-,cardId,accountId),
            pillegalcards,paccounts) ==
   cardId not in set pillegalcards and
   accountId in set dom paccounts and
   cardId in set dom paccounts(accountId).cards;
```

REQUESTING A STATEMENT

```
RequestStatement : () ==> Name * seq of Transaction *
                          nat
RequestStatement() ==
  let mk_Card(-,cardId,accountId) = curCard,
      mk_Account(cards,balance,transactions) =
            accounts(accountId)
  in
    return mk_(cards(cardId).name,
               transactions,balance)
pre curCard <> nil and cardOk and
    IsLegalCard(curCard,illegalCards,accounts)
```

ILLEGAL CARDS AND NEW ACCOUNTS

```
ReportIllegalCard : CardId ==> ()
ReportIllegalCard(cardId) ==
  illegalCards := illegalCards union {cardId};


AddAccount : AccountId * Account ==> ()
AddAccount(accountId,account) ==
  accounts := accounts munion {accountId |-> account}
pre accountId not in set dom accounts;
```

AUXILIARY FUNCTIONS

```
functions
  Encode: PinCode -> Code
  Encode(pin) ==
    pin;
-- The actual encoding procedure has not yet been chosen

  Sum: seq of real -> real
  Sum(rs) ==
    if rs = [] then 0
    else
      hd rs + Sum(tl rs)
  measure Len;

  Len: seq of real +> nat
  Len(list) == len list;
```