System Modelling and Design

A Simple ATM
Beyond Specification

Revision: 1.2, May 16, 2006

Ken Robinson

School of Computer Science & Engineering
The University of New South Wales, Sydney Australia

May 17, 2006

©Ken Robinson 2005

mailto::k.robinson@unsw.edu.au
Outline I

1 Objectives of this Lecture

2 A Simplistic Model of an ATM
   - ATM0
   - Improving the Model
   - Bank Context machine
   - Password machine
   - ATM1
   - ATM
   - Bank
   - Login
   - A Note on structure

3 Refinement
   - Implementation
   - Implementing the ATM machine
Outline III

islogged_Vvar
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.
We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance;

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the ATM0 machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the ATMO machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the \textit{ATM0} machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
A Simplistic Model of an ATM

We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the ATM0 machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the ATM0 machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
A Simplistic Model of an ATM

We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the $ATM_0$ machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
A Simplistic Model of an ATM

We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the ATM0 machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
A Simplistic Model of an ATM

We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the $ATM_0$ machine. This is likely to be the type of specification produced by someone familiar only with machine level development.
A simplistic and incorrect model of an ATM

MACHINE ATM0
SETS
ACCOUNT ;
PASSWORD ;
RESPONSE = { OK, REFUSED }
ATM0 II

VARIABLES
accounts ,
password ,
balance ,
loggedin

INVARIANT
accounts ⊆ ACCOUNT ∧
password ∈ accounts → PASSWORD ∧
balance ∈ accounts → \mathbb{N} ∧
loggedin ∈ seq ( accounts ) ∧ size ( loggedin ) ≤ 1

INITIALISATION
accounts , password := {} , {} ||
balance , loggedin := {} , []

OPERATIONS
response ← InsertCard ( account , pass ) \supseteq \\
\textbf{PRE} \quad \text{account} \in \text{ACCOUNT} \land \text{pass} \in \text{PASSWORD} \\
\textbf{THEN} \quad \text{IF} \quad \text{loggedin} = [ ] \land \\
\quad \text{account} \in \text{accounts} \land \\
\quad \text{pass} = \text{password} ( \text{account} ) \\
\quad \text{THEN} \quad \text{response} : = \text{OK} \ |
\text{loggedin} : = [ \text{account} ] \\
\text{ELSE} \quad \text{response} : = \text{REFUSED} \\
\text{END} \\
\text{END} ;
response, bal ← WithDraw (amount) ≜

PRE amount ∈ \mathbb{N} THEN

IF loggedin \neq [] THEN

LET account BE account = first (loggedin) IN

IF amount ≤ balance (account) THEN

balance (account) := balance (account) − amount ||
bal := balance (account) − amount ||
response := OK

ELSE bal ∈ \mathbb{N} || response := REFUSED

END

ELSE bal ∈ \mathbb{N} || response := REFUSED

END

END ;
response ← Deposit ( amount ) ≜

PRE amount ∈ ℤ THEN

IFloggedin ≠ [] THEN

LET account BE account = first (loggedin) IN

balance (account) := balance (account) + amount

response := OK

END

ELSE response := REFUSED

END

END ;
response, bal ← Balance ≜

IFloggedin ≠ [] THEN

LET account BE account = first (loggedin) IN

bal := balance (account) ||
response := OK

END

ELSE bal ∈ N ||
response := REFUSED

END ;
ATM0 VII

response, account ← OpenAccount ( pass ) 

PRE pass ∈ PASSWORD THEN

IF accounts ≠ ACCOUNT

THEN ANY acc WHERE acc ∈ ACCOUNT − accounts

THEN accounts := accounts ∪ { acc } ||

balance ( acc ) := 0 ||

password ( acc ) := pass ||

account := acc ||

response := OK

END

ELSE account ∈ ACCOUNT ||

response := REFUSED

END

END ;

WithDrawCard ≜loggedin := [ ]

END
Improving the Model

This \textit{ATM0} model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in \textit{BankContext} and \textit{Password}. It’s split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.
This *ATM0* model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in *BankContext* and *Password*. It's split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.
This *ATM0* model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in *BankContext* and *Password*. It’s split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.
This ATM0 model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in BankContext and Password. It’s split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.
This *ATM0* model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in *BankContext* and *Password*. It’s split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.
MACHINE  BankContext

SETS

The set of account ids

ACCOUNT ;

The set of services cards

SCARD ;

The set of responses

RESPONSE = { OK , REFUSED }
CONSTANTS

A constant, injective function, that maps accounts to service cards

GENSCARD

PROPERTIES

\[ \text{card ( ACCOUNT )} = \text{card ( SCARD )} \land \]
\[ \text{GENSCARD} \in \text{ACCOUNT} \Rightarrow \text{SCARD} \]

OPERATIONS

The operations implement the GENSCARD function and its inverse.
Bank Context machine III

\[
\text{scard} \leftarrow \text{GenScard} \ ( \text{account} ) \ \triangleq \\
\text{PRE} \quad \text{account} \in \text{ACCOUNT} \land \text{account} \in \text{dom} ( \text{GENSCARD} ) \\
\text{THEN} \quad \text{scard} := \text{GENSCARD} ( \text{account} ) \\
\text{END} ; \\
\text{account} \leftarrow \text{ExtractAccount} \ ( \text{scard} ) \ \triangleq \\
\text{PRE} \quad \text{scard} \in \text{SCARD} \land \text{scard} \in \text{ran} ( \text{GENSCARD} ) \\
\text{THEN} \quad \text{account} := \text{GENSCARD}^{-1} ( \text{scard} ) \\
\text{END}
\]

END
We are now modelling a service card, distinct from the account. We assume that the service card can be represented by information that is generated from the account, and that the account can be extracted from the service card.

Looking ahead to secure implementation to numeric operations we use the finite $SCALAR$ type from the $Scalar\_TYPE$ machine instead of $\mathbb{N}$. 
We are now modelling a service card, distinct from the account. We assume that the service card can be represented by information that is generated from the account, and that the account can be extracted from the service card.

Looking ahead to secure implementation to numeric operations we use the finite \( SCALAR \) type from the \( Scalar\_TYPE \) machine instead of \( \mathbb{N} \).
MACHINE Password
SETS PASSWORD
END
We show two stages in specifying the ATM. The first attempt, ATM1, is nearly what we are aiming for, but it contains modelling of the login management that is really nothing to do with the pure interface view of an ATM.

This machine attempts to model the observed behaviour of an ATM.

Note the extensive use of non-determinism.

MACHINE ATM1
SEES BankContext, Password, Scalar
VARIABLES loggedin
INVARIANT loggedin ⊆ ACCOUNT ∧ card (loggedin) ≤ 1
INITIALISATION loggedin := {}
response ← InsertCard ( scard, pass ) ≜

PRE scard ∈ SCARD ∧ scard ∈ ran ( GENSCARD ) ∧ pass ∈ PASSWORD THEN

IFloggedin = {} THEN

CHOICE

response := OK ||

loggedin := { GENSCARD ^−1 ( scard ) }

OR

response := REFUSED

END

ELSE response := REFUSED

END

END ;
response, money, bal ← ATMWithDraw(amount) ≜
PRE amount ∈ SCALAR THEN
IFloggedin ≠ {} THEN
  CHOICE
    response := OK || money := amount
  OR
    response := REFUSED || money := 0
END
ELSE
  response := REFUSED || money := 0
END || bal ∈ SCALAR
END ;
response, bal ← ATMDeposit(amount) ≜

PRE amount ∈ SCALAR THEN

IF loggedin ≠ {} THEN

response ∈ RESPONSE

ELSE response := REFUSED

END || bal ∈ SCALAR

END ;
\[\text{response}, \text{bal} \leftarrow \text{ATMBalance} \triangleq\]

BEGIN
  IF \( \text{loggedin} \neq \{\} \) THEN
    \(\text{response} : \in \text{RESPONSE}\)
  ELSE
    \(\text{response} := \text{REFUSED}\)
  END
 END \parallel \text{bal} : \in \text{SCALAR}

END ;
\textbf{ATM1 VI}

\texttt{\textbf{WithDrawCard} \equiv \textit{loggedin} := \{}\}

\texttt{END}
ATM is a stateless machine with nondeterministic operations.

This machine models the observed behaviour of an ATM. It does not attempt to model the reasons for the behaviour.

Note the extensive use of non-determinism.

**MACHINE ATM**

**SEES** BankContext, Password, Scalar_TYPE

**OPERATIONS**
ATM II

\[
\text{response} \leftarrow \text{InsertCard} \left( \text{scard}, \text{pass} \right) \cong \\
\text{PRE} \quad \text{scard} \in \text{SCARD} \land \text{scard} \in \text{ran} \left( \text{GENSCARD} \right) \land \\
\text{pass} \in \text{PASSWORD} \quad \text{THEN} \\
\text{response} \in \text{RESPONSE} \\
\text{END} ;
\]
response, money, bal ← ATMWithDraw(amount) ≜

PRE amount ∈ SCALAR THEN

CHOICE

response := OK || money := amount

OR

response := REFUSED || money := 0

END || bal ∈ SCALAR

END ;
\[
\text{response}, \text{bal} \leftarrow \text{ATMDeposit}(\text{amount}) \equiv \\
\text{PRE } \text{amount} \in \text{SCALAR} \text{ THEN} \\
\text{response} :\in \text{RESPONSE} \parallel \text{bal} :\in \text{SCALAR} \\
\text{END} ;
\]
$response, \; bal \leftarrow \text{ATMBalance} \triangleq$

BEGIN

$\begin{align*}
response & \in \text{RESPONSE} \quad \parallel \quad bal \in \text{SCALAR}
\end{align*}$

END ;
ATM VI

\[ \text{WithDrawCard} \triangleq \text{skip} \]

\text{END}
This machine provides a simple model of a remote banking system. The machine operations represent transactions and queries that can be requested by a client system.

```
MACHINE Bank (maxaccount)
CONSTRAINTS maxaccount ∈ 1 .. MaxScalar
SEES BankContext, Password, Scalar_TYPE, Bool_TYPE

OPERATIONS
```
response, account, scard ← OpenAccount (pass) ≜

PRE pass ∈ PASSWORD THEN

account ∈ ACCOUNT || scard ∈ SCARD ||
response ∈ RESPONSE

END ;
response ← CheckPassword ( account , pass ) ≜

PRE account ∈ ACCOUNT ∧ pass ∈ PASSWORD THEN

response :∈ RESPONSE

END ;
response, bal \leftarrow \textbf{WithDraw} (\texttt{account, amount}) \triangleq

\textbf{PRE } account \in \texttt{ACCOUNT} \land amount \in \texttt{SCALAR} \text{ THEN }

\texttt{bal} :\in \texttt{SCALAR} \parallel \texttt{response} :\in \texttt{RESPONSE}

\textbf{END} ;
response ← Deposit ( account , amount ) \equiv
\textbf{PRE} \quad \text{account} \in \text{ACCOUNT} \land \text{amount} \in \text{SCALAR} \quad \text{THEN}
\begin{align*}
\text{response} & : \in \text{RESPONSE} \\
\text{END} & ;
\end{align*}
Bank VI

\[ \text{response, bal} \leftarrow \textbf{Balance} \ (\text{account}) \ \triangleq \]
\[ \text{PRE} \quad \text{account} \in \text{ACCOUNT} \ \text{THEN} \]
\[ \text{bal} \in \text{SCALAR} \ || \ \text{response} \in \text{RESPONSE} \]
\[ \text{END} ; \]
Bank VII

\[
\begin{align*}
response & \leftarrow \text{BackUp} \triangleq response : \in BOOL; \\
response & \leftarrow \text{Restore} \triangleq response : \in BOOL
\end{align*}
\]
The login machine is responsible for maintaining login status.

**MACHINE** Login

**SEES** BankContext, Bool_TYPE

**VARIABLES** loggedin

**INVARIANT** loggedin \( \subseteq \) ACCOUNT \( \land \) card (loggedin) \( \leq \) 1

**INITIALISATION** loggedin := \{\}

**OPERATIONS**
Login II

\[
\text{login} \ (\text{account}) \triangleq \\
\text{PRE \ account} \in \text{ACCOUNT \ THEN} \\
\text{loggedin} := \{ \text{account} \} \\
\text{END} ; \\
\text{status} \leftarrow \ \text{islogged} \ \	riangleq \\
\text{status} := \text{bool} (\ \text{loggedin} \neq \{\} ) ; \\
\text{account} \leftarrow \ \text{getaccount} \ \	riangleq \\
\text{PRE \ \text{loggedin} \neq \{\} \ \THEN} \\
\text{account} \in \text{loggedin} \\
\text{END} ; \\
\text{logout} \ \	riangleq \ \text{loggedin} := \{} \\
\text{END}
\]
Notice that the ATM, Bank, Password and Login machines each SEE BankContext.

Further along the development, BankContext will be imported into the implementation of Bank. BankContext and Password will be implemented independently.
A Note on structure

Notice that the *ATM, Bank, Password* and *Login* machines each SEES *BankContext*.

Further along the development, *BankContext* will be imported into the implementation of *Bank*. *BankContext* and *Password* will be implemented independently.
A Note on structure

Notice that the *ATM*, *Bank*, *Password* and *Login* machines each SEES *BankContext*.

Further along the development, *BankContext* will be imported into the implementation of *Bank*. *BankContext* and *Password* will be implemented independently.
A Note on structure

Notice that the *ATM, Bank, Password* and *Login* machines each SEES *BankContext*.

Further along the development, *BankContext* will be imported into the implementation of *Bank*. *BankContext* and *Password* will be implemented independently.
Refinement is the name given to design in \textit{B Method (B)}. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Refinement is the name given to design in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Refinement is the name given to design in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Refinement

*Refinement* is the name given to *design* in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the *refinement relation*, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be *strengthened*, but not *weakened*. 
**Refinement** is the name given to *design* in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the *refinement relation*, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operation to be *strengthened*, but not *weakened*. 
Refinement is the name given to design in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Refinement

Refinement is the name given to design in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Refinement

*Refinement* is the name given to *design* in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the *refinement relation*, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be *strengthened*, but not *weakened*. 
**Refinement** is the name given to *design* in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the *refinement relation*, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be *strengthened*, but not *weakened*. 
Refinement is the name given to design in B. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the refinement relation, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be strengthened, but not weakened.
Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. full hiding of the variables of the imported machines is enforced, ie the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.
Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. full hiding of the variables of the imported machines is enforced, ie the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.
Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. full hiding of the variables of the imported machines is enforced, ie the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.
Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. full hiding of the variables of the imported machines is enforced, ie the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.
Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. full hiding of the variables of the imported machines is enforced, *ie* the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.
We now demonstrate that we can implement the ATM machine by importing the Bank and Login machines. The implementation is a programming exercise in which we are limited to using the operations of the imported machines. Notice that we are allowed to use sequential composition.
We now demonstrate that we can implement the ATM machine by importing the Bank and Login machines. The implementation is a programming exercise in which we are limited to using the operations of the imported machines. Notice that we are allowed to use sequential composition.
IMPLEMENTATION  \textit{ATMI} REFINES  \textit{ATM} SEES  \textit{Password}, \textit{Bool\_TYPE} IMPORTS  \\
\textit{BankContext}, \\
\textit{Bank (maxaccount)}, \\
\textit{Login} \\

OPERATIONS
response ← InsertCard ( scard, pass )

VAR bb, res, account IN

bb ← isloggedin;

IF bb = FALSE THEN

account ← ExtractAccount ( scard );

res ← CheckPassword ( account, pass );

IF res = OK THEN

login ( account )

END ;

response := res

ELSE response := REFUSED

END

END ;
response, money, bal ← ATMWithDraw(amount) ≜

VAR bb, res, acc IN

response := REFUSED; money := 0; bal := 0;
bb ← isloggedin;
IF bb = TRUE THEN
acc ← getaccount;
res, bal ← WithDraw(acc, amount);
IF res = OK THEN
money := amount
END;
response := res
END ;

END ;
response , bal ← ATMDeposit ( amount ) ≜

VAR bb , res , acc IN

bb ← isloggedin ;

IF bb = TRUE THEN

acc ← getaccount ;

response ← Deposit ( acc , amount ) ;

res , bal ← Balance ( acc )

ELSE bal := 0 ; response := REFUSED

END

END ;
response, bal ← ATMBalance \leftarrow

VAR bb, acc IN

bb ← isloggedin;

IF bb = TRUE THEN

acc ← getaccount;

response, bal ← Balance(acc)

ELSE bal := 0; response := REFUSED

END

END ;
WITHDRAWCARD \equiv \text{logout}

DEFINITIONS \quad \text{maxaccount} \equiv 100

END
In order to exercise the ATM machine it is necessary to be able to create accounts and service cards, so we extend the ATM machine by adding a clone of the OpenAccount operation.

```
MACHINE ATMBankSys
SEES BankContext, Password, Scalar_TYPE, Bool_TYPE
EXTENDS ATM

OPERATIONS
  response, account, scard ←← OpenAccount ( pass ) ≜
    PRE  pass ∈ PASSWORD THEN
      account ∈ ACCOUNT || scard ∈ SCARD || response ∈ RESPONSE
    END ;
  response ←← BackUp ≜ response ∈ BOOL ;
  response ←← Restore ≜ response ∈ BOOL
END
```
And we implement the ATMBankSys machine.

**IMPLEMENTATION** ATMBankSysI
**REFINES** ATMBankSys
**SEES** Password, Bool_TYPE
**IMPORTS**
  - BankContext,
  - Bank (maxaccount),
  - Login
**PROMOTES**
  - OpenAccount,
  - BackUp,
  - Restore

**OPERATIONS**
ATMBankSysI II

\[
\text{response} \leftarrow \quad \textbf{InsertCard} \ ( \text{scard}, \text{pass}) \equiv \\
\text{VAR} \quad \text{bb}, \text{res}, \text{account} \quad \text{IN} \\
\text{bb} \leftarrow \text{islogged} \ ; \\
\text{IF} \quad \text{bb} = \text{FALSE} \quad \text{THEN} \\
\text{account} \leftarrow \text{ExtractAccount} \ (\text{scard}) \ ; \\
\text{res} \leftarrow \text{CheckPassword} \ (\text{account}, \text{pass}) \ ; \\
\text{IF} \quad \text{res} = \text{OK} \quad \text{THEN} \\
\text{login} \ (\text{account}) \\
\text{END} \ ; \\
\text{response} := \text{res} \\
\text{ELSE} \quad \text{response} := \text{REFUSED} \\
\text{END} \\
\text{END} \ ;
\]
response, money, bal ← ATMWithDraw (amount) ≡

VAR  bb, res, acc IN

response := REFUSED; money := 0; bal := 0;

bb ← isloggedIn;
IF  bb = TRUE THEN
acc ← getaccount;
res, bal ← WithDraw (acc, amount);
IF  res = OK THEN
money := amount
END;
response := res
END
END;
ATMBankSysI IV

response, bal ← ATMDeposit(amount) ≜

VAR bb, res, acc IN

bb ← isloggedin;

IF bb = TRUE THEN

acc ← getaccount;

response ← Deposit(acc, amount);

res, bal ← Balance(acc)

ELSE bal := 0; response := REFUSED

END

END ;
response, bal ← \textbf{ATMBalance} \triangleq \\
\textbf{VAR} \quad bb, acc \textbf{ IN} \\
bb ← \textbf{isloggedin} ; \\
\textbf{IF} \quad bb = \textbf{TRUE} \quad \textbf{THEN} \\
acc ← \textbf{getaccount} ; \\
response, bal ← \textbf{Balance} ( acc ) \\
\textbf{ELSE} \quad bal := 0 ; response := \textbf{REFUSED} \\
\textbf{END} \\
\textbf{END} ;
WithDrawCard $\equiv$ logout

DEFINITIONS

$maxaccount \equiv 100$

END
In $BankR$, the refinement of $Bank$, we model the mapping from account to password with a function $accounts \rightarrow PASSWORD$.

Looking ahead to implementation, we recognise that it would be unwise to implement a mapping from account to a plaintext password. It would be more secure to encrypt the password. To provide facilities for this we introduce a new machine $Encryption$.

We also specify the operation $CheckPassword$ as comparing encrypted passwords, rather than comparing plain passwords. Notice that we need to “think ahead” on this issue: if we specified the operation as comparing plain passwords, we could not later decide to implement the operation using comparison of encrypted passwords as this is weaker than comparing plain passwords and is hence not a refinement.
In *BankR*, the refinement of *Bank*, we model the mapping from account to password with a function \( \text{accounts} \rightarrow \text{PASSWORD} \).

Looking ahead to implementation, we recognise that it would be unwise to implement a mapping from account to a plaintext password. It would be more secure to encrypt the password. To provide facilities for this we introduce a new machine *Encryption*.

We also specify the operation *CheckPassword* as comparing encrypted passwords, rather than comparing plain passwords. Notice that we need to “think ahead” on this issue: if we specified the operation as comparing plain passwords, we could not later decide to implement the operation using comparison of encrypted passwords as this is weaker than comparing plain passwords and is hence not a refinement.
In *BankR*, the refinement of *Bank*, we model the mapping from account to password with a function \( \text{accounts} \to \text{PASSWORD} \).

Looking ahead to implementation, we recognise that it would be unwise to implement a mapping from account to a plaintext password. It would be more secure to encrypt the password. To provide facilities for this we introduce a new machine *Encryption*.

We also specify the operation *CheckPassword* as comparing encrypted passwords, rather than comparing plain passwords. Notice that we need to “think ahead” on this issue: if we specified the operation as comparing plain passwords, we could not later decide to implement the operation using comparison of encrypted passwords as this is weaker than comparing plain passwords and is hence not a refinement.
In *BankR*, the refinement of *Bank*, we model the mapping from account to password with a function \( \text{accounts} \rightarrow \text{PASSWORD} \).

Looking ahead to implementation, we recognise that it would be unwise to implement a mapping from account to a plaintext password. It would be more secure to encrypt the password. To provide facilities for this we introduce a new machine *Encryption*.

We also specify the operation *CheckPassword* as comparing encrypted passwords, rather than comparing plain passwords. Notice that we need to “think ahead” on this issue: if we specified the operation as comparing plain passwords, we could not later decide to implement the operation using comparison of encrypted passwords as this is weaker than comparing plain passwords and is hence not a refinement.
Encryption I

MACHINE Encryption
SEES Password
SETS CRYPT
CONSTANTS ENCRYPT
PROPERTIES ENCRYPT ∈ PASSWORD → CRYPT

OPERATIONS
  encrypted ← Encrypt ( password ) ≜
  PRE password ∈ PASSWORD
  THEN encrypted := ENCRYPT ( password )
END
END
BankR is a refinement of Bank. This machine adds variables to resolve the non-determinism in the operations of Bank.

**REFINEMENT** BankR

**REFINES** Bank

**SEES** BankContext, Password, Encryption, Scalar_TYPE, Bool_TYPE

**VARIABLES** accounts, password, balance

**INVARIANT**

- \( \text{accounts} \subseteq \text{ACCOUNT} \land \)
- \( \text{card} ( \text{accounts} ) \leq \text{maxaccount} \land \)
- \( \text{password} \in \text{accounts} \rightarrow \text{PASSWORD} \land \)
- \( \text{balance} \in \text{accounts} \rightarrow \text{SCALAR} \)

**INITIALISATION** accounts, password, balance := {}, {}, {}

**OPERATIONS**
BankR II

\[\text{response}, \text{account}, \text{scard} \leftarrow \text{OpenAccount}(\text{pass}) \triangleq \]

**CHOICE**

**IF** \(\text{card}(\text{accounts}) \neq \text{maxaccount}\)

**THEN** ANY \(\text{acc}\) WHERE \(\text{acc} \in \text{ACCOUNT} \setminus \text{accounts}\)

**THEN** \(\text{accounts} := \text{accounts} \cup \{\text{acc}\}\) ||

\(\text{balance}(\text{acc}) := 0\) ||

\(\text{password}(\text{acc}) := \text{pass}\) ||

\(\text{account} := \text{acc}\) ||

\(\text{scard} := \text{GENSCARD}(\text{acc})\) ||

\(\text{response} := \text{OK}\)

**END**

**ELSE** \(\text{account} \in \text{ACCOUNT} \) || \(\text{scard} \in \text{SCARD}\) ||

\(\text{response} := \text{REFUSED}\)

**END**
OR

class account \in ACCOUNT \parallel class scard \in SCARD \parallel

response := REFUSED

END ;
BankR IV

response ← CheckPassword ( account, pass ) ≜

IF account ∈ accounts ∧

ENCRIPT ( password ( account ) ) = ENCRYPT ( pass )

THEN response := OK
ELSE response := REFUSED
END ;
BankR V

\[
\begin{align*}
\text{response} & , \, bal \leftarrow \text{WithDraw} \ ( \text{account} \ , \ \text{amount} ) \triangleq \\
& \quad \text{IF} \quad \text{account} \in \text{accounts} \land \text{amount} \leq \text{balance} \ ( \text{account} ) \\
& \quad \text{THEN} \quad \text{balance} \ ( \text{account} ) := \text{balance} \ ( \text{account} ) - \text{amount} \quad || \\
& \quad \quad bal := \text{balance} \ ( \text{account} ) - \text{amount} \quad || \\
& \quad \quad \text{response} := \text{OK} \\
& \quad \text{ELSE} \quad bal \in \text{SCALAR} \quad || \quad \text{response} := \text{REFUSED} \\
& \quad \text{END} ; \\
\text{response} & \leftarrow \text{Deposit} \ ( \text{account} \ , \ \text{amount} ) \triangleq \\
& \quad \text{IF} \quad \text{account} \in \text{accounts} \land \\
& \quad \quad \text{balance} \ ( \text{account} ) + \text{amount} \leq \text{MaxScalar} \quad \text{THEN} \\
& \quad \quad \text{balance} \ ( \text{account} ) := \text{balance} \ ( \text{account} ) + \text{amount} \quad || \\
& \quad \quad \text{response} := \text{OK} \\
& \quad \text{ELSE} \quad \text{response} := \text{REFUSED} \\
& \quad \text{END} ;
\end{align*}
\]
response, bal ← Balance ( account ) 

IF account ∈ accounts THEN
    bal := balance ( account ) ||
    response := OK
ELSE bal ∈ SCALAR ||
    response := REFUSED
END ;
response ← BackUp ≜
BEGIN response :∈ BOOL END ;
response ← Restore ≜
BEGIN
  ANY acc, passw, bal
  WHERE acc ⊆ ACCOUNT ∧
    card(acc) ≤ maxaccount ∧
    passw ∈ acc → PASSWORD ∧
    bal ∈ acc → SCALAR
  THEN accounts, password, balance := acc, passw, bal
END ||
response :∈ BOOL
END
END
Implementing Bank

To implement the Bank machine we will use the B-Toolkit’s base generator capability.

The database definition shown in `AccountDB base` consists of an accounts database called `accountsdb` containing records with mandatory fields `passwordf` and `balancef`.

The B-Toolkit takes the base definition and generates the machine `AccountDB`, which is imported into the implementation `BankRI`.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.
To implement the Bank machine we will use the B-Toolkit’s base generator capability.

The database definition shown in `AccountDB base` consists of an accounts database called `accountsdb` containing records with mandatory fields `passwordf` and `balancef`.

The B-Toolkit takes the base definition and generates the machine `AccountDB`, which is imported into the implementation `BankRI`.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.
To implement the Bank machine we will use the B-Toolkit’s base generator capability.

The database definition shown in *AccountDB base* consists of an accounts database called *accountsdb* containing records with mandatory fields *passwordf* and *balancef*.

The B-Toolkit takes the base definition and generates the machine *AccountDB*, which is imported into the implementation *BankRI*.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.
Implementing Bank

To implement the Bank machine we will use the B-Toolkit’s base generator capability.

The database definition shown in AccountDB base consists of an accounts database called accountsdb containing records with mandatory fields passwordf and balancef.

The B-Toolkit takes the base definition and generates the machine AccountDB, which is imported into the implementation BankRI.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.
To implement the Bank machine we will use the B-Toolkit’s base generator capability.

The database definition shown in *AccountDB base* consists of an accounts database called *accountsdb* containing records with mandatory fields *passwordf* and *balancef*.

The B-Toolkit takes the base definition and generates the machine *AccountDB*, which is imported into the implementation *BankRI*.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.
AccountDB base I

SYSTEM AccountDB
SUPPORTS BankRI
IS

BASE
  accountsdb
MANDATORY
  passwordf ∈ CRYPT ;
  balancef ∈ \mathbb{N}
END
END
MACHINE AccountDBCtx
SETS
  accountsdb_ABSOBJ
DEFINITIONS
  AccountDB_ABSOBJ ≜ accountsdb_ABSOBJ ∪ N;
  AccountDB_ABSOBJ_SETS ≜ accountsdb_ABSOBJ ∪ N;
  AccountDB_SETS ≜ N
END

MACHINE AccountDB ( max_accountsdb , CRYPT )
CONSTRAINTS
  max_accountsdb ∈ 1 . . 2147483646
SEES
  AccountDBCtx , Bool_type , Scalar_type
VARIABLES
  accountsdb ,
  locate_accountsdb ,
  passwordf ,
  balancef
INVARIANT
  accountsdb ⊆ accountsdb_ABSOBJ ∧
  card ( accountsdb ) ≤ max_accountsdb ∧
  locate_accountsdb ∈ 1 . . card ( accountsdb ) ↦ accountsdb ∧
  passwordf ∈ accountsdb → CRYPT ∧
  balancef ∈ accountsdb → SCALAR
INITIALISATION
  accountsdb := {} ∥
  locate_accountsdb := {} ∥
  passwordf := {} ∥
AccountDB II

\[ \text{balancef} := \{ \} \]

**OPERATIONS**

\[
\begin{align*}
\text{rep} & \leftarrow \text{save\_AccountDB} \triangleq \begin{aligned}
\text{BEGIN} & \; \text{rep} \in \text{BOOL} \; \text{END}; \\
\text{rep} & \leftarrow \text{restore\_AccountDB} \triangleq \\
\text{BEGIN} & \; \text{any} \\
\text{accountsdbx, locate\_accountsdbx, passwordfx, balancefx} \\
\text{WHERE} \\
\text{accountsdbx} & \subseteq \text{accountsdb\_ABSOBJ} \land \\
\text{card} \left( \text{accountsdbx} \right) & \leq \text{max\_accountsdb} \land \\
\text{locate\_accountsdbx} & \in 1 \ldots \text{card} \left( \text{accountsdbx} \right) \rightarrow \text{accountsdbx} \land \\
\text{passwordfx} & \in \text{accountsdbx} \rightarrow \text{CRYPT} \land \\
\text{balancefx} & \in \text{accountsdbx} \rightarrow \text{SCALAR} \\
\text{THEN} \\
\text{accountsdb} & := \text{accountsdbx} || \\
\text{locate\_accountsdb} & := \text{locate\_accountsdbx} || \\
\text{passwordf} & := \text{passwordfx} || \\
\text{balancef} & := \text{balancefx} \\
\text{END} || \\
\text{rep} \in \text{BOOL} \\
\text{END}; \\
\text{rep, Base\_accountsdb} & \leftarrow \text{make\_accountsdb} \left( \text{Val\_passwordf, Val\_balancef} \right) \triangleq \\
\text{PRE} \\
\text{Val\_passwordf} & \in \text{CRYPT} \land \\
\text{Val\_balancef} & \in \text{SCALAR} \land \\
\text{card} \left( \text{accountsdb} \right) & < \text{max\_accountsdb} \\
\text{THEN} \\
\text{choice} \\
\text{any} \; \text{Base\_accountsdbx, loc} \; \text{WHERE} \\
\text{Base\_accountsdbx} & \subseteq \text{accountsdb\_ABSOBJ} - \text{accountsdb} \land \\
\text{loc} & \in 1 \ldots \text{card} \left( \text{accountsdb} \right) + 1 \rightarrow \text{accountsdb} \cup \{ \text{Base\_accountsdbx} \} 
\end{aligned}
\]


AccountDB III

```
THEN
  accountsdb := accountsdb ∪ { Base_accountsdbx } ||
  passwordf ( Base_accountsdbx ) := Val_passwordf ||
  balancef ( Base_accountsdbx ) := Val_balancef ||
  Base_accountsdb := Base_accountsdbx ||
  loc_accountsdb := loc ||
  rep := TRUE
END
OR
  ANY Base_accountsdb WHERE
    Base_accountsdb ∈ accountsdb_ABSOBJ
  THEN
    Base_accountsdb := Base_accountsdbx ||
    rep := FALSE
END
END

rep ←− vld_accountsdb ( Base_accountsdb ) =
PRE Base_accountsdb ∈ accountsdb_ABSOBJ THEN
  rep := bool ( Base_accountsdb ∈ accountsdb )
END;
Base_accountsdb ←− any_accountsdb = BEGIN Base_accountsdb ∈ accountsdb_ABSOBJ END;
rep ←− eql_passwordf ( Base_accountsdb , ElemCRYPT ) =
PRE Base_accountsdb ∈ accountsdb ∧ ElemCRYPT ∈ CRYPT THEN
  rep := bool ( passwordf ( Base_accountsdb ) = ElemCRYPT )
END;
nat ←− val_balancef ( Base_accountsdb ) =
PRE Base_accountsdb ∈ accountsdb THEN
  nat := balancef ( Base_accountsdb )
END;
mod_balancef ( Base_accountsdb , nat ) =
PRE Base_accountsdb ∈ accountsdb ∧ nat ∈ SCALAR THEN
```
balancef (Base.accountsdb) ::= nat

END
IMPLEMENTATION  BankRI
REFINES  BankR
SEES
BankContext, Password, Encryption, AccountDBCtx, Scalar_TYPE, Scalar_TYPE_Ops, Bool_TYPE
IMPORTS  AccountDB ( maxaccount, CRYPT )
PROPERTIES
ACCOUNT = accountsdb_ABSOBJ ∧
SCARD = ACCOUNT
INVARIANT
accounts = accountsdb ∧
(password ; ENCRYPT ) = passwordf ∧
balance = balancef

OPERATIONS
BankRI II
response, account, scard ← OpenAccount (pass) ≜

VAR bb, db, encpass IN

account ← any_accountsdb; scard ← any_accountsdb;
response := REFUSED;
encpass ← Encrypt (pass);
bb, db ← make_accountsdb (encpass, 0);
IF bb = TRUE THEN
  account := db; scard := db; response := OK
END
END;
response ← CheckPassword( account, pass ) \equiv

VAR \quad bb, encpass IN

response := REFUSED ;

bb ← vld_accountsdb( account ) ;

IF \quad bb = TRUE THEN

encpass ← Encrypt( pass ) ;

bb ← eql_passwordf( account, encpass ) ;

IF \quad bb = TRUE THEN

response := OK

END

END

END ;
response, bal ← WithDraw ( account, amount ) ≜ 

VAR bb, balv IN 

response := REFUSED ; bal := 0 ; 
bb ← vld_accountsdb ( account ) ; 
IF bb = TRUE THEN 
balv ← val_balancef ( account ) ; 
IF amount ≤ balv THEN 
balv ← SUB ( balv, amount ) ; 
mod_balancef ( account, balv ) ; 
bal := balv ; 
response := OK 
END 
END 
END ;
response ← Deposit ( account , amount ) ≡

VAR  bb , bal  IN

response := REFUSED ;  

bb ← vld_accountsdb ( account ) ;

IF  bb = TRUE  THEN

bal ← val_balancef ( account ) ;

IF  bal ≤ MaxScalar − amount  THEN

bal ← ADD ( bal , amount ) ;

mod_balancef ( account , bal ) ;

response := OK

END

END

END ;
response, bal ← Balance(account) ⊨

VAR bb IN

bal := 0; response := REFUSED;
bb ← vld_accountsdb(account);
IF bb = TRUE THEN
  bal ← val_balancef(account);
  response := OK
END
END;
BankRI VIII

\[
\text{response} \leftarrow \text{BackUp} \triangleq \text{response} \leftarrow \text{save AccountDB} ; \\
\text{response} \leftarrow \text{Restore} \triangleq \\
\text{response} \leftarrow \text{restore AccountDB} \\
\text{END}
\]
We implement Login by importing two instances of the renameable variable machine, \textit{Rename\_Vvar}, from the standard library. These two machines implement a Boolean variable and an ACCOUNT variable, respectively, and provide a refinement of the singleton or empty set of ACCOUNT.

**IMPLEMENTATION**  \textit{LoginI}  
**REFINES**  \textit{Login}  
**SEES**  \textit{BankContext} , \textit{Bool\_TYPE}  
**IMPORTS**  \textit{loggedin\_Vvar} ( ACCOUNT ) , \textit{islogged\_Vvar} ( BOOL )  
**INVARIANT**  
\( ( \text{loggedin} \neq \{ \} \Rightarrow \text{islogged\_Vvar} = \text{TRUE} ) \land \)  
\( ( \text{loggedin} = \{ \} \Rightarrow \text{islogged\_Vvar} = \text{FALSE} ) \land \)  
\( ( \text{islogged\_Vvar} = \text{TRUE} \Rightarrow \{ \text{loggedin\_Vvar} \} = \text{loggedin} ) \)  
**INITIALISATION**  \textit{islogged\_STO\_VAR} ( FALSE )
Implementing Login II

OPERATIONS

\textbf{login} ( account ) ≜

BEGIN

\begin{align*}
\text{loggedin}_\text{STO}_\text{VAR} ( \text{account} ) ; \\
\text{islogged}_\text{STO}_\text{VAR} ( \text{TRUE} ) \\
\end{align*}

END ;

\text{status} ← \text{islogged} \cong

\begin{align*}
\text{status} ← \text{islogged}_\text{VAL}_\text{VAR} ; \\
\text{account} ← \text{getaccount} \cong \\
\text{account} ← \text{loggedin}_\text{VAL}_\text{VAR} ; \\
\text{logout} \cong \\
\text{islogged}_\text{STO}_\text{VAR} ( \text{FALSE} ) \\
\end{align*}

END
Implementing Encryption I

Having set the scene for encryption, we will implement encryption by making the encrypted password the same as the password. Clearly, we can do this without loss of generality as we could use any other encryption we wish.

IMPLEMENTATION  EncryptionI
REFINES  Encryption
SEES  Password
PROPERTIES

\[
\text{CRYPT} = \text{PASSWORD} \land \\
\text{ENCRYPT} = \text{id}(\text{PASSWORD})
\]

OPERATIONS

\[
\text{encrypted} \leftarrow \text{Encrypt}(\text{password}) \equiv \\
\text{encrypted} := \text{password}
\]

END
The following frames show the instantiation of the machines *loggedin_Vvar* and *islogged_Vvar.mch*. 
MACHINE loggedin_Vvar ( VALUE )
SEES  Bool_TYPE
VARIABLES loggedin_Vvar
INVARIANT loggedin_Vvar ∈ VALUE
INITIALISATION loggedin_Vvar :∈ VALUE

OPERATIONS

\[ \text{loggedin}_\text{VAL\_VAR} \leftarrow \text{loggedin}_\text{Vvar} \]
BEGIN
\[ \text{loggedin}_\text{VAL\_VAR} \leftarrow \text{loggedin}_\text{Vvar} \]
END ;
loggedin_STO_VAR ( vv ) ⊨
PRE
\[ vv \in VALUE \]
THEN
logedrin_Vvar II

\[ logedrin_Vvar := vv \]

\[ \text{BEGIN} \]
\[ \text{skip} \]
\[ \text{END} ; \]

\[ bb \leftarrow \text{logedrin_EQL_VAR} (vv) \equiv \]

\[ \text{PRE} \]
\[ vv \in \text{VALUE} \]

\[ \text{THEN} \]
\[ bb := \text{bool} ( logedrin_Vvar = vv ) \]

\[ \text{END} ; \]

\[ bb \leftarrow \text{logedrin_NEQ_VAR} (vv) \equiv \]

\[ \text{PRE} \]
\[ vv \in \text{VALUE} \]

\[ \text{THEN} \]
\[ bb := \text{bool} ( logedrin_Vvar \neq vv ) \]

\[ \text{END} ; \]

\[ \text{logedrin_SAV_VAR} \equiv \]

\[ \text{BEGIN} \quad \text{skip} \quad \text{END} ; \]

\[ \text{logedrin_RST_VAR} \equiv \]
BEGIN

\( \text{loggedin\_Vvar} \in VALUE \)

END ;

\( \text{loggedin\_SAVN\_VAR} \equiv \)

BEGIN skip END ;

\( \text{loggedin\_RSTN\_VAR} \equiv \)

BEGIN

\( \text{loggedin\_Vvar} \in VALUE \)

END

END
islogged_Vvar I

MACHINE islogged_Vvar ( VALUE )
SEES Bool_TYPE
VARIABLES islogged_Vvar
INVARIANT islogged_Vvar ∈ VALUE
INITIALISATION islogged_Vvar :∈ VALUE

OPERATIONS
    vv ← islogged_VAL_VAR ≜
    BEGIN
    vv := islogged_Vvar
    END ;
    islogged_STO_VAR ( vv ) ≜
    PRE
    vv ∈ VALUE
    THEN
islogged_Vvar II

\[
islogged_Vvar := vv
\]

**END ;**

\[
bb \leftarrow \text{islogged\_EQL\_VAR} ( vv ) \equiv
\]

**PRE**

\[
vv \in VALUE
\]

**THEN**

\[
bb := \text{bool} ( \text{islogged\_Vvar} = vv )
\]

**END ;**

\[
bb \leftarrow \text{islogged\_NEQ\_VAR} ( vv ) \equiv
\]

**PRE**

\[
vv \in VALUE
\]

**THEN**

\[
bb := \text{bool} ( \text{islogged\_Vvar} \neq vv )
\]

**END ;**

\[
islogged\_SAV\_VAR \equiv
\]

**BEGIN** \text{skip} **END ;**

\[
islogged\_RST\_VAR \equiv
\]
BEGIN
    islogged_Vvar :∈ VALUE
END ;

islogged_SAVN_VAR ≡ BEGIN skip END ;
islogged_RSTN_VAR ≡

BEGIN
    islogged_Vvar :∈ VALUE
END

END