The software development process with B

1. Requirement analysis
2. Specification development
   - decompose spec into meaningful components and formalize them into abstract machines
   - animate to check spec againts requirements
   - generate and prove consistency obligations
3. Design
   - identify decomposition of system implementation, including reusable components from libraries
   - create refinements of selected components
   - generate and prove refinement proof obligations
4. Coding/Integration
   - apply code generator to lowest level design

Requirement Analysis – The Banking System

1. Customers are identified by their name and the birth date
2. Customers can have any number of accounts
3. All accounts have a unique number
4. Each account has a unique owner
5. Accounts have a non-negative balance
6. ...

Abstract Machines

- A specification of a system can contain a significant amount of information. Therefore, a structural approach is needed
- In B, larger specifications can be constructed from smaller ones, permitting hierarhical specification
- The basic block of a B specification is a so called *abstract machine*
- An abstract machine is a specification of (a part of) a system using an *Abstract Machine Notation*

Abstract Machines (cont.)

- An **abstract machine** is a module encapsulating data and operations on that data. A machine is similar to a C++ object or an Ada package
- Each machine "owns" some local data and provides the essential operations to access and manipulate these data
- Variables of a machine can only be modified by operations of this machine and not by operations of other machines

Abstract Machine Notation

- Abstract Machine Notation (AMN) is the notation used to describe B abstract machines
- AMN gives B appearance and feel of a programming language, although the level of abstraction is higher
- The notation for the specification source form will be ascii. For example, `account:ACCOUNT` means that variable `account` is an element of the set (type) `ACCOUNT`
- In textbooks, graphical mathematical notation is used. The example above would look $account \in ACCOUNT$

Structure of an abstract machine

```
MACHINE(...)      machine name and parameters
CONSTRAINTS       conditions on parameters (predicate)
INCLUDES/SEES     connection to other machines (names)
SETS              local types (names)
CONSTANTS         local constants (names)
PROPERTIES        conditions on sets and constants
VARIABLES         local variables (names)
INVARIANT         invariant properties (predicate)
INITIALISATION    assignment
OPERATIONS        operations
END
```

Structure of an abstract machine (cont.)

- Note the hierarchy of constraints in the machine structure:
  - `CONSTRAINTS` constrain the machine parameters
  - `PROPERTIES` constrain the sets and constants
  - `INVARIANT` constrain the machine variables
- Constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate


```
MACHINE
  Booking

VARIABLES
  seats
INVARIANT
  seats ∈ NAT

INITIALISATION
  seats := 1000

OPERATIONS
  book = ...
  cancel = ...

END
```

MACHINE
  Name of the machine/module
  (the length of the name is always > 1)
VARIABLES (,)
  give the state of the machine
  may be changed locally in the machine
INVARIANT ($\wedge$)
  defines the types of the variables,
  defines the constraints and the relations between the variables
INITIALISATION (||)
  assignments of initial values
  should establish the invariant
OPERATIONS (;)
  relevant instructions for the user

Machine Parameters

- Machine parameters enable the specification of generic machines
- The parameters are either:
  - *sets* (upper case identifiers): denote finite non_empty sets
  - *numeric*: denote natural number constants


```
MACHINE
   Booking (max_seats)
CONSTRAINTS
   max_seats ∈ NAT ∧
   max_seats > 0

VARIABLES
   seats
INVARIANT
   seats ∈ NAT ∧
   seats ≤ max_seats
INITIALISATION
   seats := max_seats

OPERATIONS
...
```

Parameters (,)
values from outside

CONSTRAINTS (∧)
logical properties of the parameters

```
...

SETS
   RES = {ok,fail}

CONSTANTS
   Max_tickets

PROPERTIES
   Max_tickets ∈ NAT ∧
   Max_tickets = 5


...
```

SETS (;)
- local abstract types
- (modeled by sets)

CONSTANTS (,)
- read-only
- cannot occur on the left of :=

PROPERTIES (∧)
- defines the types and logical properties of the sets and constants
- can constrain accessed data


```
...

OPERATIONS
   book =
     BEGIN
        seats := seats − 1
     END;

   cancel =
     BEGIN
        seats := seats + 1
     END
END
```

Operations (;)
- change the state of the variables
- are executed in one step (i.e. no sequential composition)
- v := e is an ordinary assignment
- v,w := e,f is a multiple assignment

Assignments in "book" and "cancel" are not always possible!

We need more general operations.

```
...
OPERATIONS
  book =
    PRE    1 ≤ seats
    THEN
      seats := seats − 1
    END;

  cancel =
    PRE seats < max_seats
    THEN
      seats := seats + 1
    END
END
```

Preconditioned operation
(PRE **P** THEN **S** END)
- gives reasonable result only if the precondition is true
- if the precondition is false, we can get any result, failed execution, termination

Operation invoker should ensure that the precondition is true

Invariant and Preconditions

- The invariant of a machine is an expression of safety or integrity conditions. Satisfying the invariant should ensure the integrity and consistency of the information modelled by the state of a machine
- It is an obligation that each operation maintains the invariant: it is guranteed that the invariant is true before an operation is invoked and it is the duty of operation to ensure that the invariant is true after the operation
- The precondition of an operation should exclude all combinations of state and operation arguments that would lead to the invariant to be broken after the operation
- It is desirable that the invariant is as strong as possible, and the precondition is as weak as possible

Operations may have value and result parameters

operation

operation(value)

result $\leftarrow$ operation

result $\leftarrow$ operation(value)


```
...
OPERATIONS
   book(number) =
     PRE number ∈ NAT ∧
          number ≤ seats
     THEN
        seats := seats − number
     END;

   cancel(number) =
     PRE number ∈ NAT ∧
        seats+number ≤ max_seats
     THEN
        seats := seats + number
     END
END
```

Value parameters

- should be typed in the precondition

- the precondition states what conditions the parameters should satisfy

```
...
  res ← book(number) =
    PRE number ∈ NAT
    THEN
      IF number ≤ seats ∧
         number ≤ Max_tickets
      THEN
        seats:=seats − number ||
        res := ok
      ELSE
        res := fail
      END
    END;
  vacant ← vacant_seats =
    BEGIN vacant := seats END
END
```

Result parameters are used
- to inform outside world about the success/failure of the op
- in the enquiry operations

enquiry operations
- are used to inform the outside world about the local state
- give a result, but do not change the state (only read the state)

in the example "book" checks internally that a required condition is true and returns an error code

Operation Preconditions

- If the operation precondition contains only the type(s) of its parameters, it is called trivial. Such (*total*) operation can be invoked in any state of the machine, and for any values of it parameters
- Operations with non-trivial preconditions are *partial* operations: that is the operation may not be defined outside of the precondition
- A precondition is an assumption that the operation makes about calling environment. It is not a condition that is going to be tested by the implementor of the operation
- It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation

```
MACHINE
   Booking (max_seats)
CONSTRAINTS
   max_seats ∈ NAT ∧ max_seats > 0

SETS
   RES = {ok,fail}
CONSTANTS
   Max_tickets
PROPERTIES
   Max_tickets ∈ NAT ∧
   Max_tickets = 5

VARIABLES
   seats
INVARIANT
   seats ∈ NAT ∧ seats ≤ max_seats

INITIALISATION
   seats := max_seats
...
```



```
  res ← book(number) =
     PRE number ∈ NAT
     THEN
        IF number ≤ seats ∧
           number ≤ Max_tickets
        THEN
          seats, res := seats − number, ok
        ELSE
          res := fail
        END
     END;

  cancel(number) =
     PRE number ∈ NAT ∧
        seats+number ≤ max_seats
     THEN
        seats := seats + number
     END

  vacant ← vacant_seats =
     BEGIN vacant := seats END
END
```