Lecture aims

1. To understand the motivation, advantages, and disadvantages of design by contract.

2. To specify formal contracts for small- and medium-scale packages in SPARK.

Lecture plan

Introduction to DbC

1. Based on the metaphor of "clients" and "suppliers", who have *obligations* for and receive *benefits* from (respectively) a mutual *contract*. Contracts are formal, mathematical statements specifying module behaviour.

2. Developed by Bertrand Meyer, while a professor University of California, Santa Barbara. DbC + Eiffel efforts resulted in the 2006 ACM Software System Award.

3. Re-iterate concepts of information hiding, encapsulation, and ADTs.

Specifying ADT interfaces

1. Importance of interfaces (for information hiding of ADTs).

2. Discuss comments as interface specifications, and show example below.

3. Discuss idea of preconditions and postconditions as *structured comments*, and show 2nd example below.

4. Advantages of pre/post approach: (1) obligations are clear; and (2) forces designer to consider issues (especially preconditions and postconditions).

5. Introduce the concept of a *contract*: an agreement between two or more parties with mutual obligations and benefits. Show first table below.

6. Metaphor of contracts and software: module is the supplier and the calling code is the client. Using contracts on interfaces ensures that interaction between components is governed by contracts.

7. Contracts have at least: invariant, precondition, and postcondition.

Formal contracts

1. DbC paradigm advocates that software designers should define *formal*, *precise* and *verifiable* contracts.

2. Draw similarities with *assertions*. The DbC paradigm goes further: assertions are so valuable in establishing software correctness, they should be a routine part of the software design.

3. Show Java example below.

4. Advantages of formal contracts: precise behaviour (documentation) — thinking over the issues goes a long way towards ensuring correctness; runtime assertions; test oracles; static analysis; and no duplication of precondition checks. Disadvantages: small overhead, which is recovered quickly.

SPARK contracts

1. Pre/post annotations are optional (unlike “own” and “derives”).
2. SPARK pre/post annotations are propositions + quantifiers. All normal SPARK/Ada expressions can be used as well.
3. Show Swap_Add_Max example below. Stress importance of preconditions for Add and Divide operations
4. Things to note: `V'Old` to refer to pre-state value; similarity to using prime in Alloy; contracts specify a state machine.

```java
//Implements a bounded stack data type.
//The stack must also be less than a specified size.
public class StackImpl implements IStack
{
  //Create a new empty stack, with a maximum size, which
  // must be greater than 0
  public StackImpl(int size);

  //Add Object o to the top of the stack
  // if the stack is not full.
  public void push(Object o);

  //Return and remove the object at the top of the stack
  // if the stack is not empty.
  public Object pop();

  //Return the size of the stack.
  public int size();
}
```

	Obligation	**Benefit**
Client	Pay for product	Receives product
Supplier	Provide product	Receives income

	Obligation	**Benefit**
Client	Establish precondition	Receives result of computation
Supplier	Provide result of computation	Has precondition established

```
package Swap_Add_Max_14
  with SPARK_Mode
is
   subtype Index      is Integer range 1..100;
   type    Array_Type is array (Index) of Integer;

   procedure Swap (X, Y : in out Integer)
     with Post => (X = Y'Old and Y = X'Old);

   function Add (X, Y : Integer) return Integer
     with Pre  => (if X >= 0 and Y >= 0 then X <= Integer'Last - Y
                   elsif X < 0 and Y < 0 then X >= Integer'First - Y),
          -- The precondition may be written as X + Y in Integer if
          -- an extended arithmetic mode is selected
          Post => Add'Result = X + Y;

   function Max (X, Y : Integer) return Integer
     with Post => Max'Result = (if X >= Y then X else Y);

   function Divide (X, Y : Integer) return Integer
     with Pre  => Y /= 0 and X > Integer'First,
          Post => Divide'Result = X / Y;

   procedure Swap_Array_Elements(I, J : Index; A: in out Array_Type)
     with Post => A = A'Old'Update (I => A'Old (J),
                                    J => A'Old (I));
end Swap_Add_Max_14;
```

```
package Linear_Search
  with SPARK_Mode
is
   type Opt_Index is new Natural;

   subtype Index is Opt_Index range 1 .. Opt_Index'Last - 1;

   No_Index : constant Opt_Index := 0;

   type Ar is array (Index range <>) of Integer;

   function Search (A : Ar; I : Integer) return Opt_Index with
     Post => (if Search'Result in A'Range then A (Search'Result) = I
              else (for all K in A'Range => A (K) /= I));

end Linear_Search;
```