

Characteristics of Real-Time Systems



- **Large and Complex**
- Concurrent control of system components
- Facilities for hardware control
- Extremely reliable and safe
- Real-time facilities
- Efficiency of execution

Aim



- Review of language support for programming in the large
- Illustrate the use of modules/packages to aid decomposition and abstraction
- Separate compilation
- Modules and separate compilation in C
- Child packages and OOP in Ada 95
- OOP and Java

Decomposition and Abstraction



- Decomposition — the systematic breakdown of a complex system into smaller and smaller parts until components are isolated that can be understood and engineered by individuals and small groups
TOP DOWN DESIGN
- Abstraction — Allows detailed consideration of components to be postponed yet enables the essential part of the component to be specified
BOTTOM UP DESIGN

Modules



- A collection of logically related objects and operations
- Encapsulation — the technique of isolating a system function within a module with a precise specification of the interface
 - information hiding
 - separate compilation
 - abstract data types
- How should large systems be decomposed into modules?

The answer to this is at the heart of all Software Engineering!

Information Hiding



- A module structure supports reduced visibility by allowing information to be hidden inside its body
- The specification and body of a module can be given separately
- Ideally, the specification should be compilable without the body being written
- E.g in Ada, there is a package specification and a package body; formal relationship; compile time errors
- In C, modules are not so well formalised. Typically, programmers use a separate .h file to contain the interface to a module and a .c file for the body. No formal relationship. Errors caught at link time
- Modules are not first class language entities

Information Hiding



- Java, has the concept of a package
- There is no language syntax to represent the specification and body of a package
- A package is a directory where related classes are stored
- To add a class to the directory, simply put the package name (path name) at the beginning of the source file

Abstract data types



- A module can define both a type and the operations on the type.
- The details of the type must be hidden from the user.
- As modules are not first class, the type must be declared and instances of the type passed as a parameter to the operation.
- To ensure the user is not aware of the details of the type, it is either defined to be **private** (as in Ada) or always passed as a pointer (as you would do in C). An incomplete declaration of the type is given in the .h file.

Queue Example in Ada

```
package Queuemod is
  type Queue is limited private;
  procedure Create (Q : in out Queue);
  function Empty (Q : Queue) return Boolean;
  procedure Insert (Q : in out Queue; E : Element);
  procedure Remove (Q : in out Queue; E : out Element);
private
  -- none of the following declarations are externally visible
  type Queuenode;
  type Queueptr is access Queuenode;
  type Queuenode is
    record
      Contents : Processid; Next : Queueptr;
    end record;
  type Queue is
    record
      Front : Queueptr; Back : Queueptr;
    end record;
end Queuemod;
```

Queue Example in C (Header File)

```
typedef struct queue_t *queue_ptr_t;
```

```
queue_ptr_t create();
```

```
int empty(queue_ptr_t Q);
```

```
void insertE(queue_ptr_t Q, element E);
```

```
void removeE(queue_ptr_t Q, element *E);
```

Object-Oriented Programming



- OOP has:
 - type extensibility (**inheritance**)
 - automatic object initialisation (**constructors**)
 - automatic object finalisation (**destructors**)
 - run-time dispatching of operations (**polymorphism**)
- Ada 95 supports the above through tagged types and class-wide programming
- Java supports OOP through the use of classes

OOP and Ada

- Based on type extensions (**tagged types**) and dynamic polymorphism (**class-wide types**)

```
type A is record ... end record;  -- normal record type
```

```
type EA is tagged record ... end record;  -- tagged type
```

```
procedure Op1(E : EA; Other_Param : Param);
```

```
    -- primitive operation
```

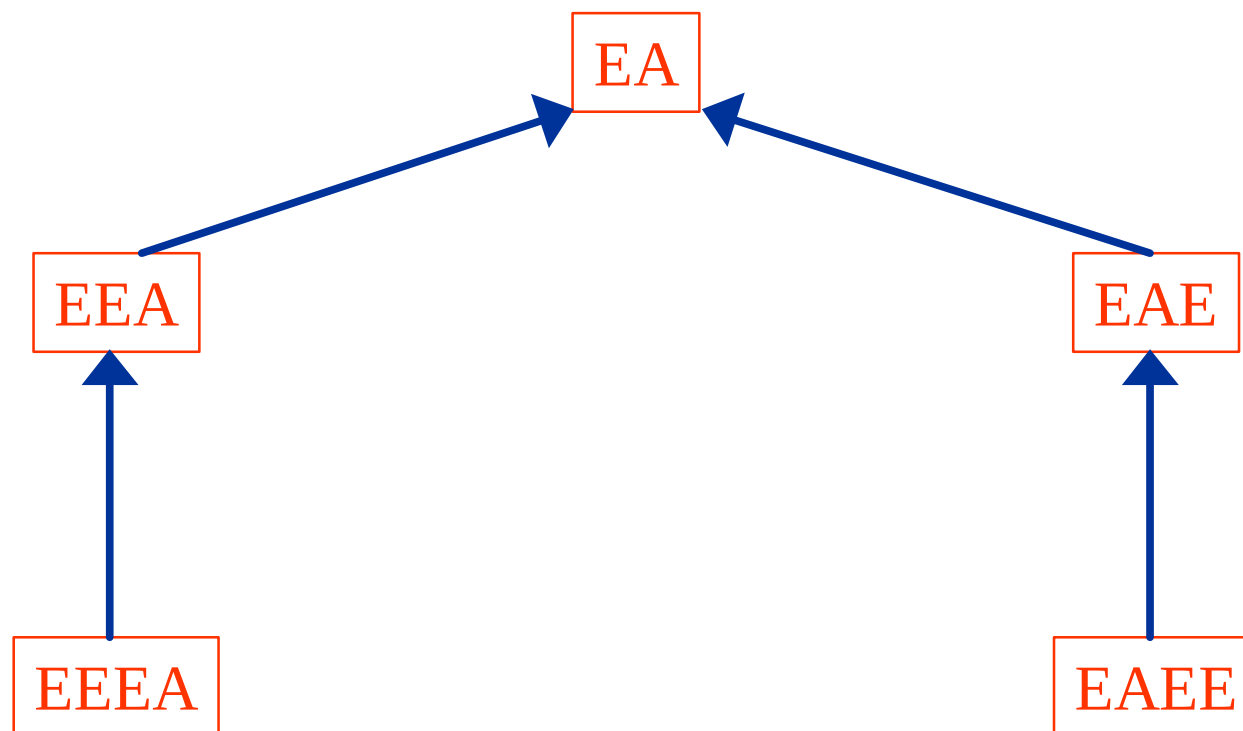
```
procedure Op2(E : EA; Other_Param : Param);
```

```
    -- primitive operation
```

Ada and OOP

```
type EEA is new EA with record ... end record;  
-- inherit OP1  
  
procedure Op2(E : EEA; Other_Param : Param);  
-- override Op2  
  
procedure Op3(E : EEA; Other_Param : Param);  
-- add new primitive operation  
  
type EEEA is new EA with record ... end record;  
...  
type EAE is new EA with record ... end record;  
...  
type EAEE is new EAE with record ... end record;  
...
```

Ada and OOP



Type Hierarchy routed at EA
called EA'Class

Class-wide Programming



```
procedure Generic_Call(X : EA'Class) is  
begin  
    OP1(X, Param) ;  
end Generic_Call;
```

Results in run-time dispatching

Child Packages

```
package Coordinate_Class is
  type Coordinates is tagged private;

  procedure Plot(P: Coordinates);

  procedure Set_X(P: Coordinates; X: Float);
  function Get_X(P: Coordinates) return Float;
  -- similarly for Y
private
  type Coordinates is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Objects;
```

Child Packages

```
package Coordinate_Class.Three_D is
  type Three_D is new Coordinates with private;
  -- new primitive operations
  procedure Set_Z(P: Coordinates; Z: Float);
  function Get_Z(P: Coordinates) return Float;
  procedure Plot(P: Three_D); -- overrides the Plot subprogram
private
  type Three_D is new Coordinates with
    record
      Z : Float;
    end record;
end Coordinate_Class.Three_D;
```

- Allow access to parent's private data without going through the parent's interface
- Reduces recompilation

Controlled Types

- With these types, it is possible to define subprograms that are called (automatically) when objects of the type:
 - are created (**initialize**)
 - cease to exist (**finalize**)
 - are assigned a new value (**adjust**)
- To gain access to these features, the type must be derived from **Controlled**, a predefined type declared in the library package **Ada.Finalization**
- This defines procedures for **Initialize**, **Finalize** and **Adjust**
- When a type is derived from **Controlled**, these procedures may be overridden

OOP and Java



- Based on the class construct
- Each class encapsulates data (**instance variables**) and operations on the data (**methods** including **constructor** methods)
- Each class can belong to a package
- It may be local to the package or visible to other packages (in which case it is labelled **public**)
- Other class modifiers are **abstract** and **final**
- Similarly, methods and instance variables have modifiers as being
 - **public** (visible outside the class)
 - **protected** (visible only within package or in a subclass)
 - **private** (visible only to the class)

Java Example

```
import somepackage.Element; // import element type
package queues; // package name

class QueueNode // class local to package
{
    Element data;
    QueueNode next;
}

public class Queue // class available from outside the package
{
    QueueNode front, back; // instance variables
    public Queue() // public constructor
    {
        front = null;
        back = null;
    }
}
```

Java Example

```
public void insert(Element E) // visible method
{
    QueueNode newNode = new QueueNode();
    newNode.data = E;  newNode.next = null;
    if(empty()) {front = newNode;}
    else { back.next = newNode; }
    back = newNode;
}

public Element remove() //visible method
{
    if(!empty()) { Element tmpE = front.data;
        front = front.next; if(empty()) back = null; }
    // garbage collection will free up the QueueNode object
    return tmpE;
}

public boolean empty() // visible method
{ return (front == null); }
}
```

Inheritance and Java

```
package coordinate;
public class Coordinate // Java is case sensitive
{
    float X, Y;

    public Coordinate(float initial_X, float initial_Y) // constructor
    { X = initial_X;
      Y = initial_Y; };

    public void set(float F1, float F2)
    { X = F1;
      Y = F2; };

    public float getX()
    { return X; }

    public float getY()
    { return Y; };

    public void plot() {
        // plot a two D point};
};
```

Inheritance and Java

```
package coordinate;
public class ThreeDimension extends Coordinate {
    // subclass of Coordinate

    float Z; // new field

    public ThreeDimension(float initialX, float initialY,
        float initialZ) // constructor
    { super(initialX, initialY); // call superclass constructor
      Z = initialZ;
    };

    public void set(float F1, float F2, float F3) //overridden method
    { super.set(F1, F2); // call superclass set
      Z = F3;
    };

    public float getZ() // new method
    { return Z;}

    public void plot() {//overridden method
        // plot a three D point};
};
```

Inheritance and Java

- Unlike Ada, **all** method calls are dispatching

```
{  
    Coordinate A = new Coordinate(0f, 0f);  
    A.plot();  
}
```

would plot a two dimension coordinate; where as

```
{  
    Coordinate A = new Coordinate(0f, 0f);  
    ThreeDimension B = new ThreeDimension(0f, 0f, 0f);  
  
    A = B;  
    A.plot();  
}
```

will plot a three D coordinate even though A was originally declared to be of type `Coordinate`. This is because A and B are reference types. By assigning B to A only the reference has changed not the object itself.

The Object Class

- All classes are implicit subclasses of the Object class

```
public class Object {  
    ...  
    public boolean equals(Object obj);  
  
    // methods to support monitors  
    public final void wait() throws IllegalMonitorStateException,  
        InterruptedException;  
    public final void wait(long millis) throws  
        IllegalMonitorStateException, InterruptedException;  
    public final void wait(long millis, int nanos) throws  
        IllegalMonitorStateException, InterruptedException;  
    public final void notify() throws IllegalMonitorStateException;  
    public final void notifyAll() throws IllegalMonitorStateException;  
  
    //override for finalization  
    protected void finalize()  
        throws Throwable;  
}
```

Interfaces in Java



- Interfaces in Java augment classes to increase the reusability of code (compare with Ada's generics)
- An interface is a special form of class that defines the specification of a set of methods and constants
- They are by definition abstract so no instances of interfaces can be declared
- Instead, one or more classes can implement an interface, and objects implementing interfaces can be passed as arguments to methods by defining the parameter to be of the interface type
- Interfaces allow relationships to be constructed between classes outside of the class hierarchy

Interface Example

```
package interfaceExamples;
```

```
public interface Ordered {  
    boolean lessThan (Ordered o);  
};
```

- lessThan takes as a parameter any object that implements the Ordered interface

Interface Example

```
import interfaceExamples.*;
class ComplexNumber implements Ordered {
    protected float realPart;
    protected float imagPart;

    public boolean lessThan(Ordered O) // interface implementation
    {
        ComplexNumber CN = (ComplexNumber) O; // cast the parameter
        if((realPart*realPart + imagPart*imagPart) <
            (CN.getReal()*CN.getReal() + CN.getImag()*CN.getImag()))
        { return true; }
        return false;
    };

    public ComplexNumber (float I, float J) // constructor
    { realPart = I; imagPart = J; };

    public float getReal() { return realPart;};
    public float getImag() { return imagPart; };
}
```

Interface Example

```
package interfaceExamples;
public class ArraySort
{
    public static void sort (Ordered oa[], int size) //sort method
    {
        Ordered tmp;
        int pos;

        for (int i = 0; i < size - 1; i++) {
            pos = i;
            for (int j = i + 1; j < size; j++) {
                if (oa[j].lessThan(oa[pos])) {
                    pos = j;
                }
            }
            tmp = oa[pos];
            oa[pos] = oa[i];
            oa[i] = tmp;
        };
    };
};
```

Interface Example

```
public static Ordered largest(Ordered oa[], int size)
// largest method
{
    Ordered tmp;
    int pos;

    pos = 0;
    for (int i = 1; i < size; i++) {
        if (! oa[i].lessThan(oa[pos])) {
            pos = i;
        };
    };
    return oa[pos];
};
}
```

Interface Example

```
{  
  
    ArraySort AR = new ArraySort();  
    ComplexNumber arrayComplex[] = { // say  
        new ComplexNumber(6f,1f),  
        new ComplexNumber(1f, 1f),  
        new ComplexNumber(3f,1f),  
        new ComplexNumber(1f, 0f),  
        new ComplexNumber(7f,1f),  
        new ComplexNumber(1f, 8f),  
        new ComplexNumber(10f,1f),  
        new ComplexNumber(1f, 7f)  
    };  
    // array unsorted  
    AR.sort(arrayComplex, 8);  
    // array sorted  
}
```

Summary



- Module supports: information hiding, separate compilation and abstract data types
- Ada and C have a static module structure
- C informally supports modules; Java has a dynamic module structure called a class
- Both packages in Ada (and Java) and classes in Java have well-defined specifications which act as the interface between the module and the rest of the program
- Separate compilation enables libraries of precompiled components to be constructed
- The decomposition of a large program into modules is the essence of programming in the large
- The use of abstract data types or object-oriented programming, provides one of the main tools programmers can use to manage large software systems