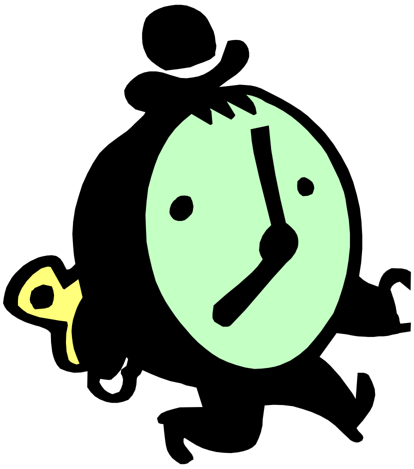
Real-Time Systems Programming

Summer-Semester 2002
Lecture 9
16 May 2002



Programming in the Large

The 5-Minute Review Session

1) Does *assembler* programming matter in the world of real-time programming?
2) What are good *criteria* for designing a real-time programming language?
3) What were the stated *goals* for the design of Ada?
4) What facilities do Ada/C/Java provide for representing *real numbers*?
5) What types of *real-time facilities* do we distinguish?
6) How does a computer tell the *time*?

Overview

1) Modules – information hiding
2) Separate compilation
3) Abstract data types
4) Object-oriented programming
5) Reusability

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

- ***Acknowledgment:*** This lecture is based in part on the slides kindly provided by the companion web site to [Burns and Wellings 2001]

Programming in the Large

Algorithms + Data Structures = Programs.

Niklaus Wirth

- Real-time software systems are typically too complex to fit this schema

Algorithms + Data Structures = Modules.

[Burns and Wellings 2001]

- In the following, want to explore how Ada/Java/C support this paradigm

Recall Architectural Design

- ***Abstraction:***
  - Allows to postpone detailed consideration of components
  - Yet can specify essential part of the component
  - ***BOTTOM UP DESIGN***
- ***Decomposition:***
  - Systematic breakdown of a complex system into smaller and smaller parts, or ***modules***
  - Until components are isolated that can be understood and engineered by individuals and small groups
  - ***TOP DOWN DESIGN***

Modules

- ... are 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
- Can separate *specification* and *body* of a module
- Ideally, can compile specification without knowing body
- Modules are (typically) not first class language entities

Information Hiding

- ***Ada:***
  - Have package specification and a package body
  - Formal relationship
  - Errors are caught at compile time
- ***Java:***
  - Has the concept of a *package*
    - A directory where related classes are stored
    - No language syntax to represent the specification and body of a package
  - To add a class to the directory, simply put the package name (path name) at the beginning of the source file

Information Hiding

- ***C:***
  - Modules are not so well formalised
  - Typically, programmers use
    - a .h file to contain the interface to a module and
    - a .c file for the body
  - No formal relationship
  - Errors caught at link time

Separate Compilation

- ... is particularly desirable if program is constructed from modules
- ... allows programmers to concentrate on current module and still construct – at least in part – a complete program
  - Can *test* separately
  - Can check logical consistency across program
  - Saves *resources*
- Also applies to *library programming*

Separate Compilation

- ***Ada:***
  - Module specification and body are seen as distinct entities of library
  - Can also provide "stub" for later inclusion (**`is separate`**)
- ***C:***
  - Can include header files

Abstract Data Types

- ***Recall:*** *Data types*
  - Allow programs to manipulate objects abstracted from implementation
  - Can increase robustness via compile-time consistency checking (*type checking*)
- *Taking this concept further:*
  - Allow the user to define additional types and operations on them
  - These are called ***Abstract Data Types*** (ADTs)

Abstract Data Types

- A module defines a *type* and the *operations* on the type
- Want to hide details of the type from user
- *A complication:*
  - Want to allow separate compilation of module specification and its body
  - However, compiler needs to know *size* of type!
- The solution in ...
  - ***C:*** indirection (use pointers, of known size)
  - ***Java:*** indirection (passing by reference)
  - ***Ada:*** define type as **private**

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;
```


- **`limited private`** means that only the subprograms defined in this package can be applied to the type
- A limited private type is therefore a true abstract data type (ADT)
- If a type is declared just **`private`**, then, in addition to the defined subprograms, assignment and equality test are available to the user

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);
```

- This .h file contains an incomplete specification

Object-Oriented Programming

- ADTs by themselves *not* sufficient for OOP
- OOP has:
  - Type extensibility (inheritance)
  - Run-time dispatching of operations (polymorphism)
  - Automatic object initialisation (constructors)
  - Automatic object finalisation (destructors)
- Ada 95 supports the above through *tagged types* and *class-wide* programming
- Java supports OOP though the use of *classes*

OOP in Ada

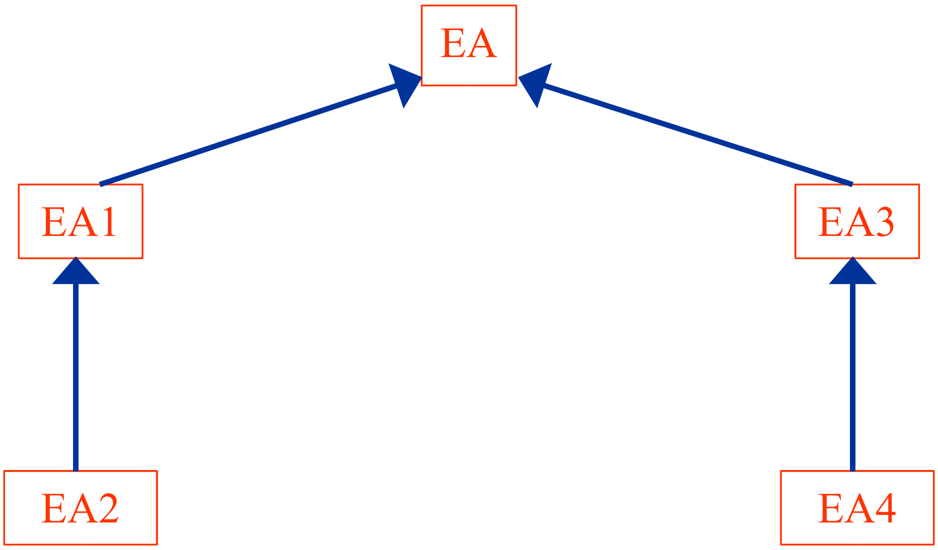
- Type extensions (tagged types)
- Dynamic polymorphism (class-wide types)

```
-- Normal record type
type A is record … end record;

-- Tagged type
type EA is tagged record … end record;

-- Primitive operations
procedure Op1(E : EA; Other_Param : Param);
procedure Op2(E : EA; Other_Param : Param);
```

OOP in Ada

```
-- Inherit OP1
type EA1 is new EA with record … end record;

-- Override Op2
procedure Op2(E : EA1; Other_Param : Param);

-- Add new primitive operation
procedure Op3(E : EA1; Other_Param : Param);


type EA2 is new EA1 with record … end record;
...

type EA3 is new EA with record … end record;
...

type EA4 is new EA3 with record … end record;
...
```

OOP in Ada



Type hierarchy (family) rooted at EA
is called **`EA'Class`**

- ***Note:***
  - Ada supports only inheritance from a single parent
  - However, can achieve multiple inheritance using the generic facilities of the language

Ada: Class-wide Programming

- ... allows to manipulate *families* of types

```
procedure Generic_Plot(P : Coordinates'Class) is
begin
     -- Do some house keeping
   Plot(P);
     -- Call the Plot procedure defined for type
     -- of actual value of P
end Generic_Plot;
```

- *This results in run-time dispatching*
- ***Pro:*** convenience, abstraction
- ***Con:*** lack of predictability!

Ada: Child Packages

- ***Problem 1:*** If a package is changed, all clients of that package must be recompiled
  - This contradicts OO's desire to facilitate *incremental* development
- ***Problem 2:*** Access to private types can only be made in body of package
  - Hence, extending private tagged types requires further language facilities
- ***The solution: Child Packages***
  - Allow access to parent’s private data without going through the parent’s interface
  - Reduce recompilation

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;

  -- Overrides the Plot subprogram
  procedure Plot(P: Three_D);

private
  type Three_D is new Coordinates with
  record
    Z : Float;
  end record;
end Coordinate_Class.Three_D;
```

Controlled Types

- ... allow to define subprograms that are called (automatically) when objects of the type
  - are created (initialization)
  - cease to exist (finalization)
  - are assigned a new value (adjustment)

Ada: Controlled Types

- To gain access to these features:
  - Type must be derived from `Controlled`
  - This is a predefined type declared in the library package `Ada.Finalization`
- This defines procedures for
  - `Initialize`
  - `Finalize`
  - `Adjust`
- When a type is derived from `Controlled`, these procedures may be overridden

OOP and Java

- ***Recall:*** OO facilities may be based on
  - *Type extensions* (Oberon, Ada)
  - Introduction of *classes* into language (Java)
- Each class encapsulates
  - Data (***instance variables***)
  - Operations on the data (***methods*** including ***constructor*** methods)
- Each class can belong to a ***package***

- A class may be
  - Local to the package or
  - Visible to other packages (in which case it is labelled `public`)
- Other class modifiers:
  - `abstract` (cannot create objects from this directly)
  - `final` (cannot derive subclasses)
- 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

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

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

// Class available from outside the package
public class Queue
{
  QueueNode front, back;    // instance variables

  public Queue()            // public constructor
  {
    front = null;
    back  = null;
  }
```

Java Example

```java
  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 void remove(Element E) // visible method
  {
    if (!empty()) { E = front.data;
      front = front.next; }
  }  // Garbage collection will free up the QueueNode object

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


- ***Note*:**
  - There is no concept of a *package specification* as in Ada
  - However, similar functionality can be provided using *interfaces* – see later

Inheritance and Java

- Inheritance in Java is obtained by deriving one class from another
- As Ada, Java supports only inheritance from a single parent
- However, can achieve *multiple inheritance* using *interfaces* (see later)

Inheritance and Java

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

  // Constructor
  public Coordinate(float initial_X, float initial_Y)
  { 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;

// Introduce a subclass of Coordinate
public class ThreeDimension extends Coordinate
{
  float Z;              // new field

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

Inheritance and Java

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

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

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

Inheritance and Java

- Unlike Ada, *all* method calls are dispatching
  - ... with the associated timing unpredictability

```
Coordinate A = new Coordinate(0f, 0f);
A.plot();    // Plots a 2-D coordinate

ThreeDimension B = new ThreeDimension(0f, 0f, 0f);

A = B;       // Recall: A and B are reference types
A.plot();    // Plots a 3-D coordinate
```

The Object Class

- All classes are implicit subclasses of class **Object**

```
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;
}
```


- Most of these methods will be discussed further in the context of monitors
- There are further methods not listed here:
  - `getClass()`
  - `toString()`
  - `hashCode()`
  - `clone()`

Reusability

- SW production is an expensive business – and costs are still rising
- ***One reason:***
  - SW is typically constructed "from scratch"
  - Compare this with the situation in HW!
- Obtaining ***SW reuse*** is a quest of SW engineering
  - *However, apart from specific areas (e.g., numerical analysis), this quest is still unfulfilled!*
- ***One obstacle:***
  - Strong typing

Interfaces in Java

- ... augment classes to increase the ***reusability*** of code
- Are similar to Ada's generics
- Are a special form of class that defines the specification of a set of methods and constants
- Allow relationships to be constructed between classes outside of the class hierarchy
- Are by definition *abstract*
  - No instances of interfaces can be declared
  - Instead, one or more classes can *implement* an interface
  - Objects implementing interfaces can be passed as arguments to methods by defining the parameter to be of the interface type

Java: Interface Example

```
package interfaceExamples;

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

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

Java: Interface Example

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

  // Interface implementation
  public boolean lessThan(Ordered O)
  {
    // Cast the parameter
    ComplexNumber CN = (ComplexNumber) O;

    if((realPart*realPart + imagPart*imagPart) <
       (CN.getReal()*CN.getReal() +
        CN.getImag()*CN.getImag()))
    { return true; }
    return false;
  };
```

Java: Interface Example

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

    public float getReal() {
      return realPart;
    };

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

Java: Interface Example

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


- Note that when two objects are exchanged, their *reference values* are exchanged
  - Hence, the type of object does not matter
  - Only prerequisite: must implement `Ordered` interface

Java: Interface Example

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

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

Java: Interface Example

```
{
  ArraySort AR = new ArraySort();

  // Create some (unsorted) array
  ComplexNumber arrayComplex[] = {
    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)
  };

  // Now sort array
  AR.sort(arrayComplex, 8);
}
```

Summary I

- Modules support:
  - Information hiding
  - Separate compilation
  - Abstract data types
- Ada and C have a *static* module structure.
  - C only 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 interface between module and rest of program

Summary II

- 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

Summary III

- *Strong typing*
  - is generally particularly desirable for real-time systems due to the robustness it provides
  - ***but*** is an impediment to SW *reuse*
- *Java* offers the *interface* mechanism to circumvent this problem
- Ada (and C++) provide *generic* primitive to enhance reuse

To Go Further

- ***Programming in the large***
  - ☞ [Burns and Wellings 2001] – Chapter 4