Real-Time Systems Programming

Summer-Semester 2002
Lecture 10
17 May 2002

Programming in the Large

Dependability Terminology

Where are we?

1) ***Programming in the large***
   - ***Object-oriented programming***
   - ***Reusability***
2) Dependability terminology

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

```
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

```
  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

- *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

Where are we?

1) Programming in the large

2) ***Dependability terminology***

- ***Reliability***
- ***Safety***
- ***Maintainability***
- ***Availability***
- ***Security***

Dependability Requirements

- ***Dependability***: The metafunctional attributes of a system that relate to the quality of service to its users during an extended interval of time
  - Reliability
  - Safety
  - Maintainability
  - Availability
  - Security
- Dependability is often *the* critical aspect of real-time systems!

Reliability

- ***Given:*** System operational at time t
- ***Reliability*** of system:
  - Probability ***R(T)*** that system will provide specified service throughout an interval $[t, t + T]$
- ***Failure rate***
  - Expected number $\lambda(T)$ of failures of the system for a time interval T

$$R(T) = e^{-\lambda(T)}$$

- ***Mean Time to Failure*** (***MTTF***): $1/\lambda$
- ***Ultrahigh reliability***: typically $MTTF > 10^9$ hrs

Safety

- ***Malign (critical) failure mode***:
  - "Cost" of failure exceeds utility of system during normal operation by orders of magnitude
  - Airbags, airtraffic control, nuclear power plants, ...
- ***Benign failure mode***:
  - Uncritical failures
- ***Safety***:
  - Reliability regarding critical failure modes
- ***Typical:***
  - Need ultra-high reliability
  - No single component failure may lead to critical system failure (e.g., TÜV)

Maintainability

- ***Given:*** System with benign failure at time t
- ***Maintainability***:
  - Probability $M(T)$ that system is repaired within $[t, t + T]$
- ***Repair rate***:
  - Expected number $\mu(T)$ of repairs of the system for a time interval T

$$M(T) = e^{-\mu(T)}$$

- ***Mean Time to Repair*** (***MTTR***): $1/\mu$
- There is often a conflict between reliability and maintainability
  - ***Example***: hardware modularisation

Availability

- ***Mean Time Between Failures*** (***MTBF***)

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

- ***Availability***:
  - Probability *A* that a system will provide specified service
- For systems with constant λ and μ:

$$A = \text{MTTF}/\text{MTBF}$$

  - Can increase A by increasing MTTF or by decreasing MTTR – or both

Downtime

- Availability corresponds to certain ***downtime***

Availability	Downtime/year	Example Component
90%	> 1 month	Unattended PC
99%	≈ 4 days	Maintained PC
99,9%	≈ 9 hrs	Cluster
99,99%	≈ 1 hr	Multicomputer
99,999%	≈ 5 mins	Embedded System (PC hw)
99,9999%	≈ 30 secs	ES (special hw)

[Veríssimo and Rodrigues 2001]

Security

- ***Security***:
  - Ability to prevent unauthorized access to information or services
  - Confidentiality, privacy
  - Theft, fraud
- Traditionally an issue for database/transaction systems
- Increasingly relevant for embedded systems as well (message interception/alteration, property protection)
- Difficult to quantify

Summary

- The ***Dependability*** of a system is given by its
  - ***Reliability:*** Measure of continuous delivery of correct service
  - ***Safety:*** Reliability wrt critical failures
  - ***Maintainability:*** Measure of time to restoration of correct service
  - ***Availability:*** Reliability + Maintainability
  - ***Security:*** Ability to prevent unauthorized access to information or services

To Go Further

- ***Dependability:***
  - ☞ [Veríssimo and Rodrigues 2001], Chapter 6
  - ☞ [Kopetz 1997], Chapter 6
  - ☞ [Burns and Wellings 2001], Chapter 5
- ***Precise, Widely-Used Terminology on Dependability:***
  - ☞ Laprie, J. C. (Ed.), *Dependability: Basic Concepts and Terminology*, Springer, 1992 (Working Group 10.4 on Fault-Tolerant Computing of the International Federation of Information Processing (IFIP))
  - ☞ Terms are defined in English, French, German, Japanese