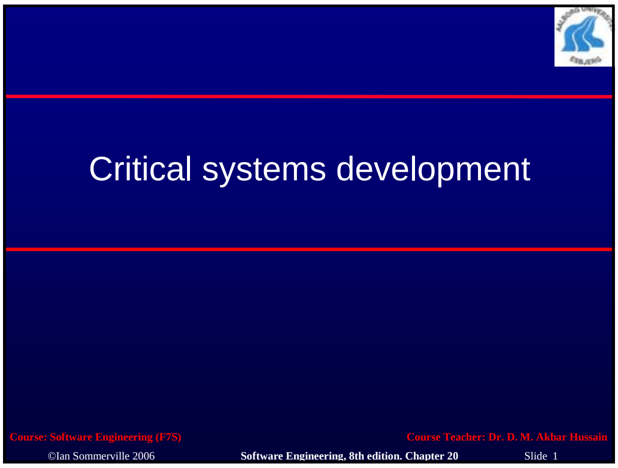
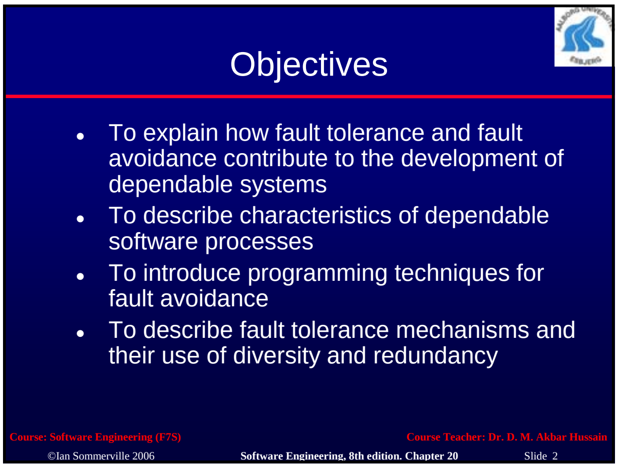
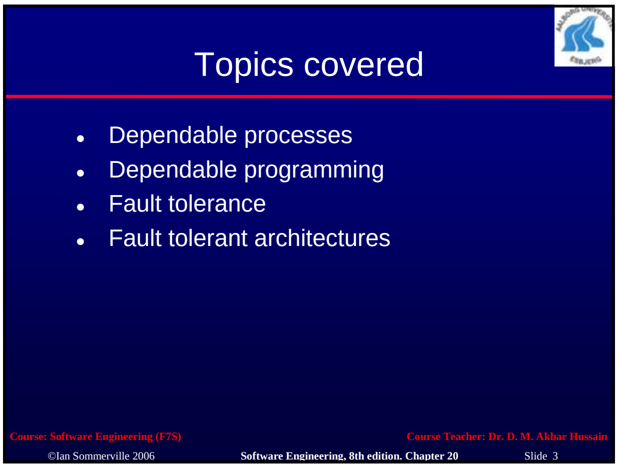


Critical systems development



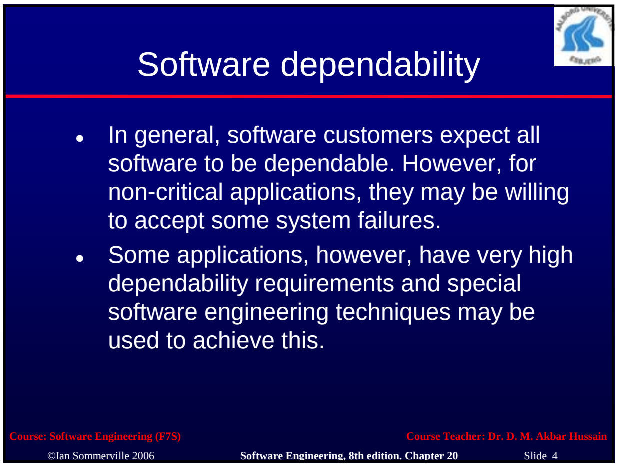
Objectives

- To explain how fault tolerance and fault avoidance contribute to the development of dependable systems
- To describe characteristics of dependable software processes
- To introduce programming techniques for fault avoidance
- To describe fault tolerance mechanisms and their use of diversity and redundancy



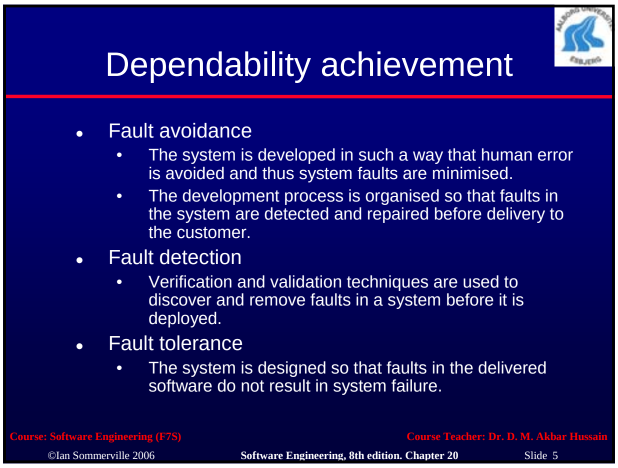
Topics covered

- Dependable processes
- Dependable programming
- Fault tolerance
- Fault tolerant architectures



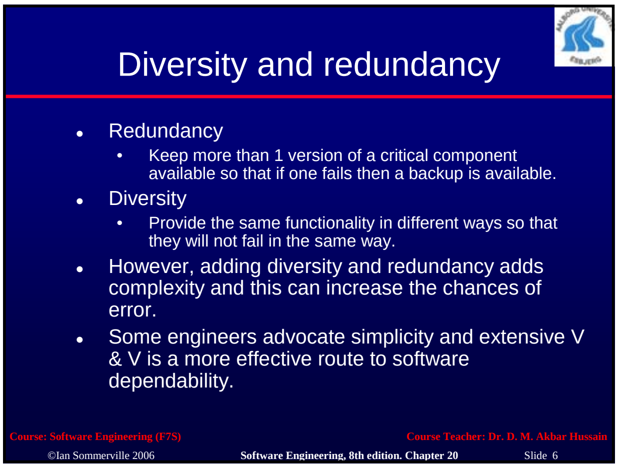
Software dependability

- In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures.
- Some applications, however, have very high dependability requirements and special software engineering techniques may be used to achieve this.



Dependability achievement

- Fault avoidance
  - The system is developed in such a way that human error is avoided and thus system faults are minimised.
  - The development process is organised so that faults in the system are detected and repaired before delivery to the customer.
- Fault detection
  - Verification and validation techniques are used to discover and remove faults in a system before it is deployed.
- Fault tolerance
  - The system is designed so that faults in the delivered software do not result in system failure.



Diversity and redundancy

- Redundancy
  - Keep more than 1 version of a critical component available so that if one fails then a backup is available.
- Diversity
  - Provide the same functionality in different ways so that they will not fail in the same way.
- However, adding diversity and redundancy adds complexity and this can increase the chances of error.
- Some engineers advocate simplicity and extensive V & V is a more effective route to software dependability.

Diversity and redundancy examples

- Redundancy. Where availability is critical (e.g. in e-commerce systems), companies normally keep backup servers and switch to these automatically if failure occurs.
- Diversity. To provide resilience against external attacks, different servers may be implemented using different operating systems (e.g. Windows and Linux)

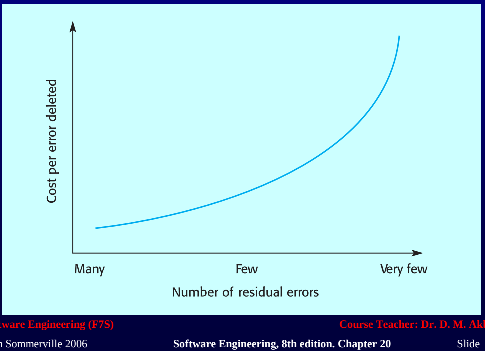
Fault-free software

- Current methods of software engineering now allow for the production of fault-free software, at least for relatively small systems.
- Fault-free software means software which conforms to its specification. It does NOT mean software which will always perform correctly as there may be specification errors.
- The cost of producing fault free software is very high. It is only cost-effective in exceptional situations. It is often cheaper to accept software faults and pay for their consequences than to expend resources on developing fault-free software.

Fault-free software development

- Dependable software processes
- Quality management
- Formal specification
- Static verification
- Strong typing
- Safe programming
- Protected information

Fault removal costs



Dependable processes

- To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process.
- A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people.
- For fault detection, it is clear that the process activities should include significant effort devoted to verification and validation.

Dependable process characteristics

Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Standardised	A comprehensive set of software development standards that define how the software is to be produced and documented should be available.
Auditable	The process should be understandable by people apart from process participants who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Robust	The process should be able to recover from failures of individual process activities.

Validation activities

- Requirements inspections.
- Requirements management.
- Model checking.
- Design and code inspection.
- Static analysis.
- Test planning and management.
- Configuration management, discussed in Chapter 29, is also essential.

Dependable programming

- Use programming constructs and techniques that contribute to fault avoidance and fault tolerance
  - Design for simplicity;
  - Protect information from unauthorised access;
  - Minimise the use of unsafe programming constructs.

Information protection

- Information should only be exposed to those parts of the program which need to access it. This involves the creation of objects or abstract data types that maintain state and that provide operations on that state.
- This avoids faults for three reasons:
  - the probability of accidental corruption of information is reduced;
  - the information is surrounded by 'firewalls' so that problems are less likely to spread to other parts of the program;
  - as all information is localised, you are less likely to make errors and reviewers are more likely to find errors.

A queue specification in Java

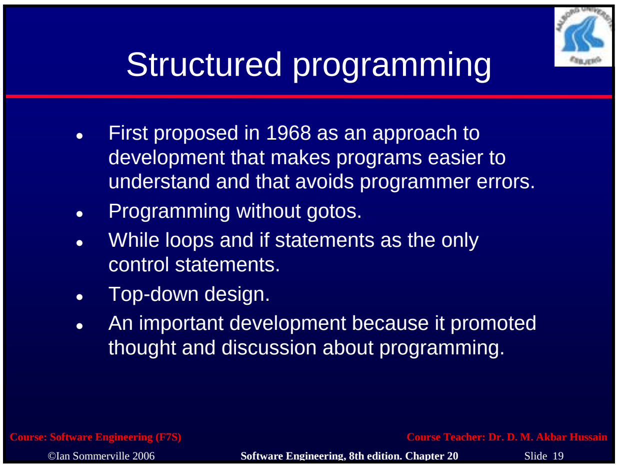
```
interface Queue {

    public void put (Object o) ;
    public void remove (Object o) ;
    public int size () ;

} //Queue
```

Signal declaration in Java

```
class Signal {

    static public final int red = 1 ;
    static public final int amber = 2 ;
    static public final int green = 3 ;

    public int sigState ;
}
```

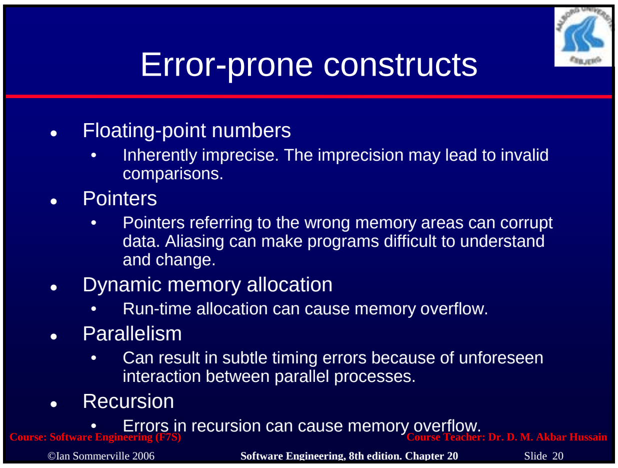
Safe programming

- Faults in programs are usually a consequence of programmers making mistakes.
- These mistakes occur because people lose track of the relationships between program variables.
- Some programming constructs are more error-prone than others so avoiding their use reduces programmer mistakes.



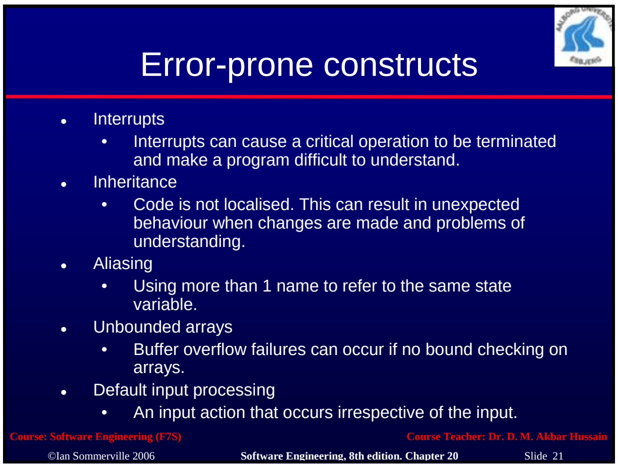
Structured programming

- First proposed in 1968 as an approach to development that makes programs easier to understand and that avoids programmer errors.
- Programming without gotos.
- While loops and if statements as the only control statements.
- Top-down design.
- An important development because it promoted thought and discussion about programming.



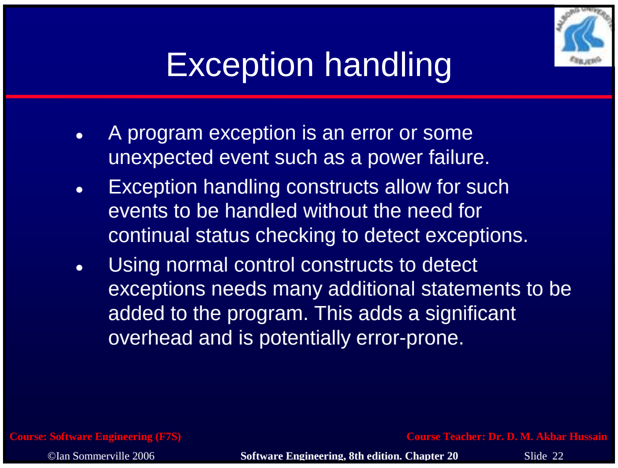
Error-prone constructs

- Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
- Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- Dynamic memory allocation
  - Run-time allocation can cause memory overflow.
- Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- Recursion
  - Errors in recursion can cause memory overflow.



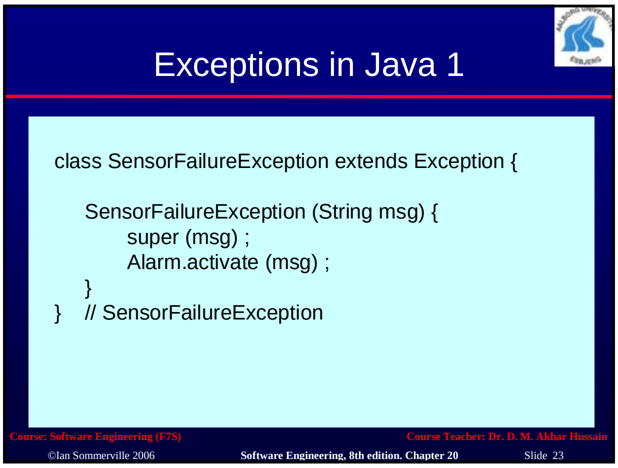
Error-prone constructs

- Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
- Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding.
- Aliasing
  - Using more than 1 name to refer to the same state variable.
- Unbounded arrays
  - Buffer overflow failures can occur if no bound checking on arrays.
- Default input processing
  - An input action that occurs irrespective of the input.



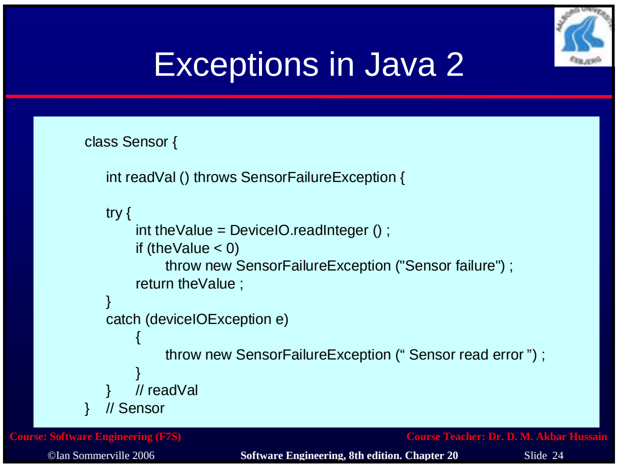
Exception handling

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.



Exceptions in Java 1

```
class SensorFailureException extends Exception {

    SensorFailureException (String msg) {
          super (msg) ;
          Alarm.activate (msg) ;
    }
}   // SensorFailureException
```



Exceptions in Java 2

```
class Sensor {

    int readVal () throws SensorFailureException {

    try {
        int theValue = DeviceIO.readInteger () ;
        if (theValue < 0)
            throw new SensorFailureException ("Sensor failure") ;
        return theValue ;
    }
    catch (deviceIOException e)
        {
            throw new SensorFailureException (“ Sensor read error ”) ;
        }
    }     // readVal
}   // Sensor
```

A temperature controller

- Exceptions can be used as a normal programming technique and not just as a way of recovering from faults.
- Consider an example of a freezer controller that keeps the freezer temperature within a specified range.
- Switches a refrigerant pump on and off.
- Sets off an alarm is the maximum allowed temperature is exceeded.
- Uses exceptions as a normal programming technique.

Freezer controller 1

```
class FreezerController {
    Sensor tempSensor = new Sensor () ;
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal () ;
    final float dangerTemp = (float) -18.0 ;
    final long coolingTime = (long) 200000.0 ;
    public void run ( ) throws InterruptedException {
    try { Pump.switchIt (Pump.on) ;
          do {
                if (freezerTemp > tempDial.setting ())
                      if (Pump.status == Pump.off)
                      {     Pump.switchIt (Pump.on) ;
                            Thread.sleep (coolingTime) ;
                                    }
```

Freezer controller 2

```
                if (freezerTemp > dangerTemp)
                     throw new FreezerTooHotException () ;
                freezerTemp = tempSensor.readVal () ;
           } while (true) ;

      } // try block
      catch (FreezerTooHotException f)
      {     Alarm.activate ( ) ; }
      catch (InterruptedException e)
      {
           System.out.println (“Thread exception”) ;
           throw new InterruptedException ( ) ;
      }
  } //run
  } // FreezerController
```

Fault tolerance

- In critical situations, software systems must be fault tolerant.
- Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- Fault tolerance means that the system can continue in operation in spite of software failure.
- Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.

Fault tolerance actions

- Fault detection
  - The system must detect that a fault (an incorrect system state) has occurred.
- Damage assessment
  - The parts of the system state affected by the fault must be detected.
- Fault recovery
  - The system must restore its state to a known safe state.
- Fault repair
  - The system may be modified to prevent recurrence of the fault. As many software faults are transitory, this is often unnecessary.

Fault detection and damage assessment

- The first stage of fault tolerance is to detect that a fault (an erroneous system state) has occurred or will occur.
- Fault detection involves defining constraints that must hold for all legal states and checking the state against these constraints.

Insulin pump state constraints

```
// The dose of insulin to be delivered must always be greater
// than zero and less that some defined maximum single dose

insulin_dose >= 0 & insulin_dose <= insulin_reservoir_contents

// The total amount of insulin delivered in a day must be less
// than or equal to a defined daily maximum dose

cumulative_dose <= maximum_daily_dose
```

Fault detection

- Preventative fault detection
  - The fault detection mechanism is initiated before the state change is committed. If an erroneous state is detected, the change is not made.
- Retrospective fault detection
  - The fault detection mechanism is initiated after the system state has been changed. This is used when a incorrect sequence of correct actions leads to an erroneous state or when preventative fault detection involves too much overhead.

Type system extension

- Preventative fault detection really involves extending the type system by including additional constraints as part of the type definition.
- These constraints are implemented by defining basic operations within a class definition.

PositiveEvenInteger 1

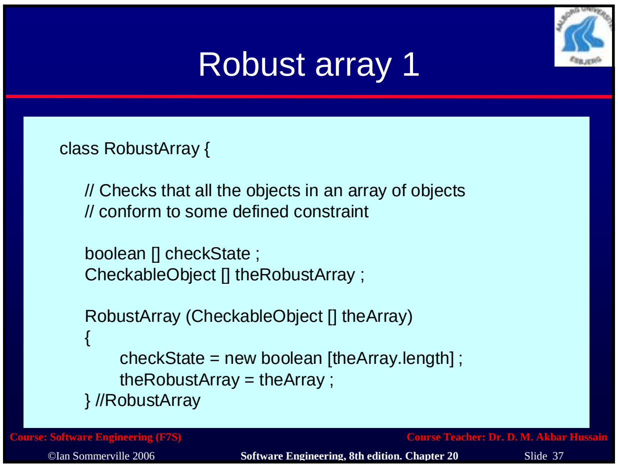
```
class PositiveEvenInteger {

    int val = 0 ;

    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;

    }// PositiveEvenInteger
```

PositiveEvenInteger 2

```
    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException ();
        else
            val = n ;
    } // assign

    int toInteger ()
    {
        return val ;
    } //to Integer

    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val) ;
    } // equals

} //PositiveEven
```

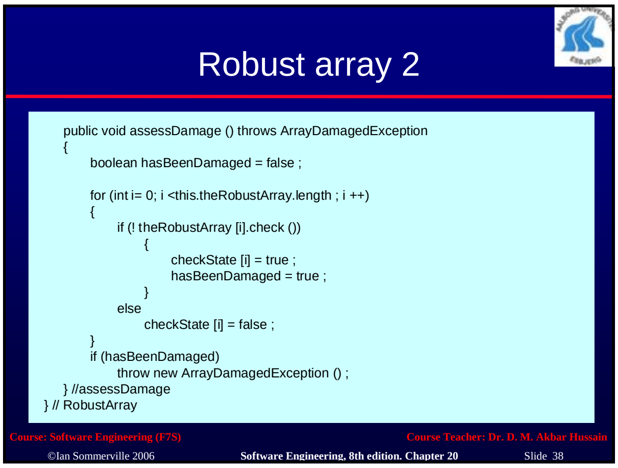
Damage assessment

- Analyse system state to judge the extent of corruption caused by a system failure.
- The assessment must check what parts of the state space have been affected by the failure.
- Generally based on ‘validity functions’ that can be applied to the state elements to assess if their value is within an allowed range.



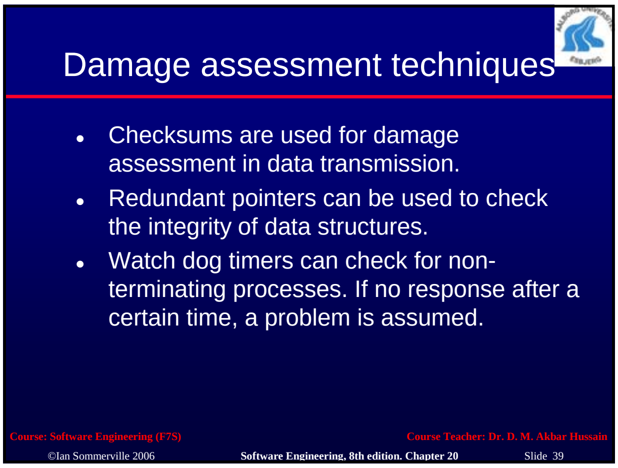
Robust array 1

```
class RobustArray {

    // Checks that all the objects in an array of objects
    // conform to some defined constraint

    boolean [] checkState ;
    CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray
```



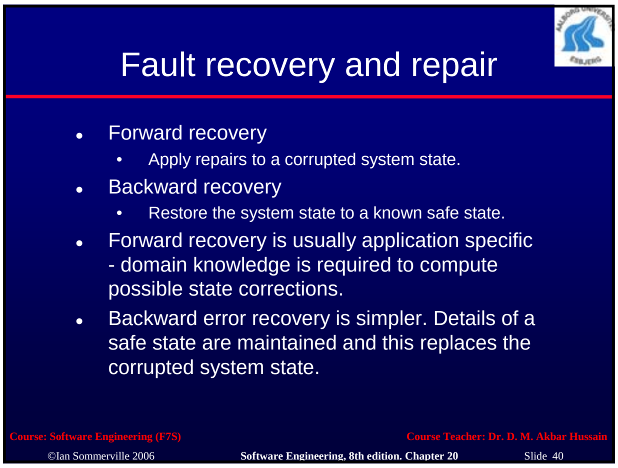
Robust array 2

```
    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
                {
                    checkState [i] = true ;
                    hasBeenDamaged = true ;
                }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException () ;
    } //assessDamage
} // RobustArray
```



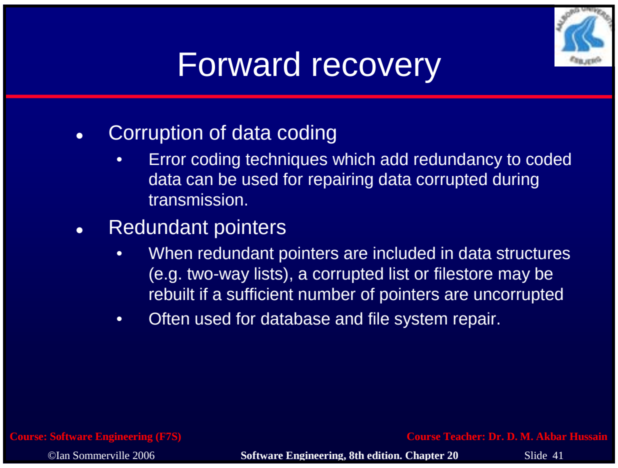
Damage assessment techniques

- Checksums are used for damage assessment in data transmission.
- Redundant pointers can be used to check the integrity of data structures.
- Watch dog timers can check for non-terminating processes. If no response after a certain time, a problem is assumed.



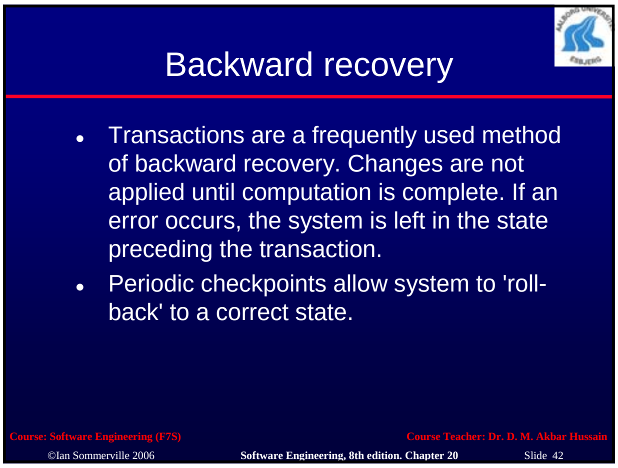
Fault recovery and repair

- Forward recovery
  - Apply repairs to a corrupted system state.
- Backward recovery
  - Restore the system state to a known safe state.
- Forward recovery is usually application specific - domain knowledge is required to compute possible state corrections.
- Backward error recovery is simpler. Details of a safe state are maintained and this replaces the corrupted system state.



Forward recovery

- Corruption of data coding
  - Error coding techniques which add redundancy to coded data can be used for repairing data corrupted during transmission.
- Redundant pointers
  - When redundant pointers are included in data structures (e.g. two-way lists), a corrupted list or filestore may be rebuilt if a sufficient number of pointers are uncorrupted
  - Often used for database and file system repair.



Backward recovery

- Transactions are a frequently used method of backward recovery. Changes are not applied until computation is complete. If an error occurs, the system is left in the state preceding the transaction.
- Periodic checkpoints allow system to 'roll-back' to a correct state.

Safe sort procedure

- A sort operation monitors its own execution and assesses if the sort has been correctly executed.
- It maintains a copy of its input so that if an error occurs, the input is not corrupted.
- Based on identifying and handling exceptions.
- Possible in this case as the condition for a'valid' sort is known. However, in many cases it is difficult to write validity checks.

Safe sort 1

```
class SafeSort {

    static void sort ( int [] in tarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];

        // copy the input array

        for (int i = 0; i < intarray.length ; i++)
            copy [i] = intarray [i] ;
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
```

Safe sort 2

```
            if (order == Sort.ascending)
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError () ;
            else
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError () ;
        } // try block
        catch (SortError e )
        {
            for (int i = 0; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Array not sorted") ;
        } //catch
    } // sort
} // SafeSort
```

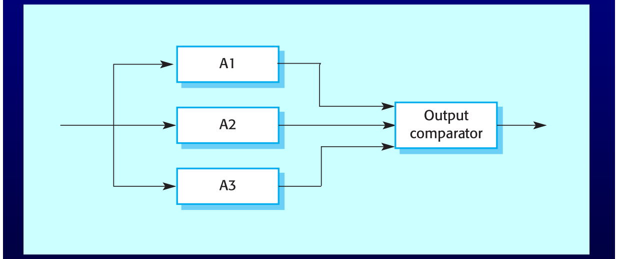
Fault tolerant architecture

- Defensive programming cannot cope with faults that involve interactions between the hardware and the software.
- Misunderstandings of the requirements may mean that checks and the associated code are incorrect.
- Where systems have high availability requirements, a specific architecture designed to support fault tolerance may be required.
- This must tolerate both hardware and software failure.

Hardware fault tolerance

- Depends on triple-modular redundancy (TMR).
- There are three replicated identical components that receive the same input and whose outputs are compared.
- If one output is different, it is ignored and component failure is assumed.
- Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure.

Hardware reliability with TMR



Output selection

- The output comparator is a (relatively) simple hardware unit.
- It compares its input signals and, if one is different from the others, it rejects it. Essentially, the selection of the actual output depends on the majority vote.
- The output comparator is connected to a fault management unit that can either try to repair the faulty unit or take it out of service.

Fault tolerant software architectures

- The success of TMR at providing fault tolerance is based on two fundamental assumptions
  - The hardware components do not include common design faults;
  - Components fail randomly and there is a low probability of simultaneous component failure.
- Neither of these assumptions are true for software
  - It isn't possible simply to replicate the same component as they would have common design faults;
  - Simultaneous component failure is therefore virtually inevitable.
- Software systems must therefore be diverse.

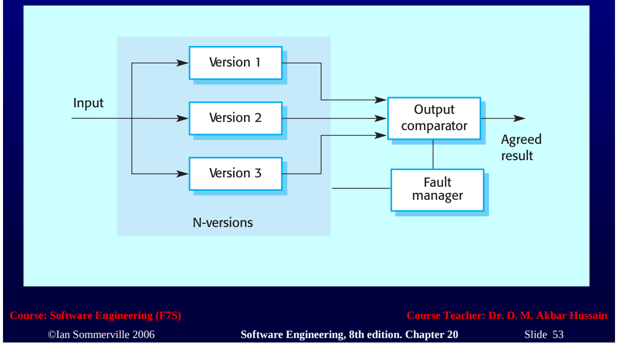
Design diversity

- Different versions of the system are designed and implemented in different ways. They therefore ought to have different failure modes.
- Different approaches to design (e.g object-oriented and function oriented)
  - Implementation in different programming languages;
  - Use of different tools and development environments;
  - Use of different algorithms in the implementation.

Software analogies to TMR

- N-version programming
  - The same specification is implemented in a number of different versions by different teams. All versions compute simultaneously and the majority output is selected using a voting system.
  - This is the most commonly used approach e.g. in many models of the Airbus commercial aircraft.
- Recovery blocks
  - A number of **explicitly** different versions of the same specification are written and executed in sequence.
  - An acceptance test is used to select the output to be transmitted.

N-version programming



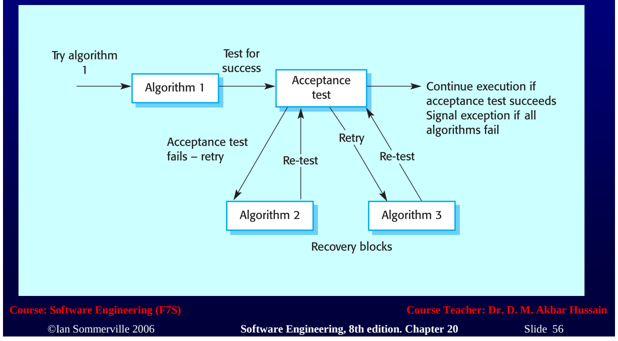
Output comparison

- As in hardware systems, the output comparator is a simple piece of software that uses a voting mechanism to select the output.
- In real-time systems, there may be a requirement that the results from the different versions are all produced within a certain time frame.

N-version programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
- There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Recovery blocks


Try algorithm 1
Algorithm 1
Test for success
Acceptance test
Continue execution if acceptance test succeeds Signal exception if all algorithms fail
Acceptance test fails – retry
Re-test
Retry
Re-test
Algorithm 2
Algorithm 3

Recovery blocks

Recovery blocks

- These force a different algorithm to be used for each version so they reduce the probability of common errors.
- However, the design of the acceptance test is difficult as it must be independent of the computation used.
- There are problems with this approach for real-time systems because of the sequential operation of the redundant versions.

Problems with design diversity

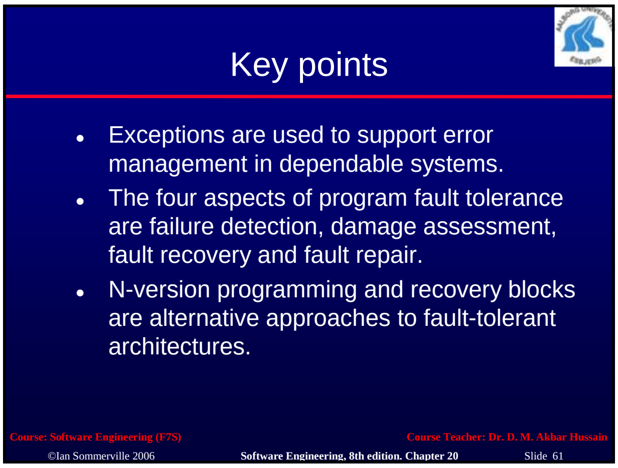
- Teams are not culturally diverse so they tend to tackle problems in the same way.
- Characteristic errors
  - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;
  - Specification errors;
  - If there is an error in the specification then this is reflected in all implementations;
  - This can be addressed to some extent by using multiple specification representations.

Specification dependency

- Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
- This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate.
- This has been addressed in some cases by developing separate software specifications from the same user specification.

Key points

- Dependability in a system can be achieved through fault avoidance, fault detection and fault tolerance.
- The use of redundancy and diversity is essential to the development of dependable systems.
- The use of a well-defined repeatable process is important if faults in a system are to be minimised.
- Some programming constructs are inherently error-prone - their use should be avoided wherever possible.



Key points

- Exceptions are used to support error management in dependable systems.
- The four aspects of program fault tolerance are failure detection, damage assessment, fault recovery and fault repair.
- N-version programming and recovery blocks are alternative approaches to fault-tolerant architectures.