SWEN 6301 Software Construction

Lecture 4: Creating High-Quality Code

Ahmed Tamrawi


Software Construction
a very complicated task

Creating High-Quality Code

Design in Construction | Working Classes | High Quality Routines | Defensive Programming

High Quality Routines

What is a routine?

A **routine** is an individual method or procedure invocable for a single purpose. Examples include a function in C++, a method in Java, a function or sub procedure in Microsoft Visual Basic

What is a high-quality routine? **That's a harder question**.

What is a high-quality routine?

C++ Example of a Low-Quality Routine

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

What is a high-quality routine?

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
          profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

The routine has a bad name. `HandleStuff()` tells you nothing about what the routine does.

The routine isn't documented.

The routine has a bad layout. The physical organization of the code on the page gives few hints about its logical organization.

The routine's input variable, `inputRec`, is changed. If it's an input variable, its value should not be modified

The routine reads and writes global variables—it reads from `corpExpense` and writes to `profit` . It should communicate with other routines more directly than by reading and writing global variables.

The routine doesn't have a single purpose. It initializes some variables, writes to a database, does some calculations—none of which seem to be related to each other in any way. A routine should have a single, clearly defined purpose.

The routine doesn't defend itself against bad data. If `crntQtr` equals 0 , the expression `ytdRevenue` * 4.0 / (double) `crntQtr` causes a divide-by-zero error.

The routine uses several magic numbers: 100 , 4.0, 12 , 2 , and 3 .

Some of the routine's parameters are unused: `screenX` and `screenY` are not referenced within the routine.

The routine has too many parameters. The upper limit for an understandable number of parameters is about 7

The routine's parameters are poorly ordered and are not documented

One of the routine's parameters is passed incorrectly: `prevColor` is labeled as a reference parameter (&) even though it isn't assigned a value within the routine.

Valid Reasons to Create a Routine

Reduce Complexity

The single most important reason to create a routine is to reduce a program's complexity. Create a routine to hide information so that you won't need to think about it.

Other reasons to create routines: minimizing code size and improving maintainability and correctness

But without the abstractive power of routines, complex programs would be impossible to manage.

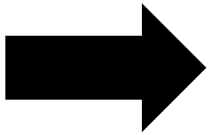
An indication that a routine needs to be broken out is loop deep nesting or a conditional

Valid Reasons to Create a Routine

Introduce an intermediate, understandable abstraction

Putting a section of code into a well-named routine is one of the best ways to document its purpose

```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```



```
leafName = GetLeafName( node )
```

Valid Reasons to Create a Routine

Avoid duplicate code

Undoubtedly the most popular reason for creating a routine is to avoid duplicate code.

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
   sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
   sum_b += array_b[i];

int average_b = sum_b / 4;
```

Valid Reasons to Create a Routine

Hide Sequences

It's a good idea to hide the order in which events happen to be processed

For example, a sequence might be found when you have two lines of code that read the top of a stack and decrement a *stackTop*variable.

Put those two lines of code into a *PopStack()* routine to hide the assumption about the order in which the two operations must be performed

Hiding that assumption will be better than baking it into code from one end of the system to the other.

Valid Reasons to Create a Routine

Hide Pointer Operations

Pointer operations tend to be hard to read and error prone. By isolating them in routines, you can concentrate on the intent of the operation rather than on the mechanics of pointer manipulation

if the operations are done in only one place, you can be more certain that the code is correct. If you find a better data type than pointers, you can change the program without traumatizing the code that would have used the pointers.

```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

Valid Reasons to Create a Routine

Improve portability

Use of routines isolates nonportable capabilities, explicitly identifying and isolating future portability work.

Nonportable capabilities include nonstandard language features, hardware dependencies, operating-system dependencies, and so on.

Valid Reasons to Create a Routine

Simplify Complicated Boolean Tests

Understanding complicated boolean tests in detail is rarely necessary for understanding program flow.

Putting such a test into a function makes the code more readable because (1) the details of the test are out of the way and (2) a descriptive function name summarizes the purpose of the test.

Giving the test a function of its own emphasizes its significance. It encourages extra effort to make the details of the test readable inside its function.

Valid Reasons to Create a Routine

Improve Performance

You can optimize the code in one place instead of in several places.

Centralizing code into a routine means that a single optimization benefits all the code that uses that routine, whether it uses it directly or indirectly.

Having code in one place makes it practical to recode the routine with a more efficient algorithm or in a faster, more efficient language.

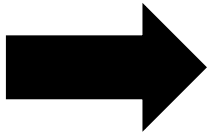
Operations That Seem Too Simple to Put Into Routines

Constructing a whole routine to contain two or three lines of code might seem like overkill, but experience shows how helpful a good small routine can be.

Small routines offer several advantages. One is that they improve readability.

Pseudocode Example of a Calculation

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```



Pseudocode Example of a Function Call to a Calculation Function

```
points = DeviceUnitsToPoints( deviceUnits )
```

Pseudocode Example of a Calculation Converted to a Function

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

Operations That Seem Too Simple to Put Into Routines

Pseudocode Example of a Calculation That Expands Under Maintenance

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
   if ( DeviceUnitsPerInch() <> 0 )
      DeviceUnitsToPoints = deviceUnits *
         ( POINTS_PER_INCH / DeviceUnitsPerInch() )
   else
      DeviceUnitsToPoints = 0
   end if
End Function
```

If that original line of code had still been in a dozen places, the test would have been repeated a dozen times, for a total of 36 new lines of code. A simple routine reduced the 36 new lines to 3.

Design at the Routine Level

Cohesion

how closely the operations in a routine are related

Coupling

the relationships between functions

Design at the Routine Level: **Cohesion**

Some programmers prefer the term "strength"; how strongly related are the operations in a routine

`Cosine()`

A function like *Cosine()* is perfectly cohesive because the whole routine is dedicated to performing one function.

`CosineAndTan()`

A function like CosineAndTan() has lower cohesion because it tries to do more than one thing. The goal is to have each routine do one thing well and not do anything else.

One study of 450 routines found that 50 percent of the highly cohesive routines were fault free, whereas only 18 percent of routines with low cohesion were fault free
(Card, Church, and Agresti1986)

Another study of a different 450 routines (which is just an unusual coincidence) found that routines with the highest coupling-to-cohesion ratios had 7 times as many errors as those with the lowest coupling-to-cohesion ratios and were 20 times as costly to fix
(Selby and Basili1991)

Design at the Routine Level: *Desired* **Cohesion**

Functional Cohesion

Functional cohesion is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation

- Compute Cosine of Angle
- Verify Alphabetic Syntax
- Read Transaction Record
- Determine Customer Mortgage Repayment
- Compute Point of Impact of Missile
- Calculate Net Employee Salary
- Assign Seat to Airline Customer

Design at the Routine Level: *Acceptable* **Cohesion**

Sequential Cohesion

Sequential cohesion exists when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together.

For example, given a birth date, calculates an employee's age and time to retirement.

If the routine calculates the age and then uses that result to calculate the employee's time to retirement, it has sequential cohesion.

Design at the Routine Level: *Acceptable* **Cohesion**

Communicational Cohesion

Communicational cohesion occurs when operations in a routine make use of the same data and aren't related in any other way.

For example, suppose you wrote a function to query a database to get the name and office number for an employee in your company.

It may make sense for your application, but the only common point between the two operations is that the data comes from the same employee record.

a. Find Title of Book
b. Find Price of Book
c. Find Publisher of Book
d. Find Author of Book

Design at the Routine Level: *Acceptable* **Cohesion**

Temporal Cohesion

Temporal cohesion occurs when operations are combined into a routine because they are all done at the same time.

Some programmers consider temporal cohesion to be unacceptable because it's sometimes associated with bad programming practices such as having a mixture of dissimilar code in a Startup() routine.

To avoid this problem, think of temporal routines as organizers of other events.

have the *temporally cohesive* routine call other routines to perform specific activities rather than performing the operations directly itself. **But this raises the issue of *choosing a name that describes the routine at the right level of abstraction***

a. Put out Milk Bottles
b. Put out Cat
c. Turn off TV
d. Brush Teeth

It will be clear that the point of the routine is to orchestrate activities rather than to do them directly.

Design at the Routine Level: *Unacceptable* **Cohesion**

Procedural Cohesion

Procedural cohesion occurs when operations in a routine are done in a specified order.

The routine has procedural cohesion because it puts a set of operations in a specified order and the operations don't need to be combined for any other reason.

To achieve better cohesion, put the separate operations into their own routines.

- Clean Utensils from Previous Meal
- Prepare Chicken for Roasting
- Make Phone Call
- Take Shower
- Chop Vegetables
- Set Table

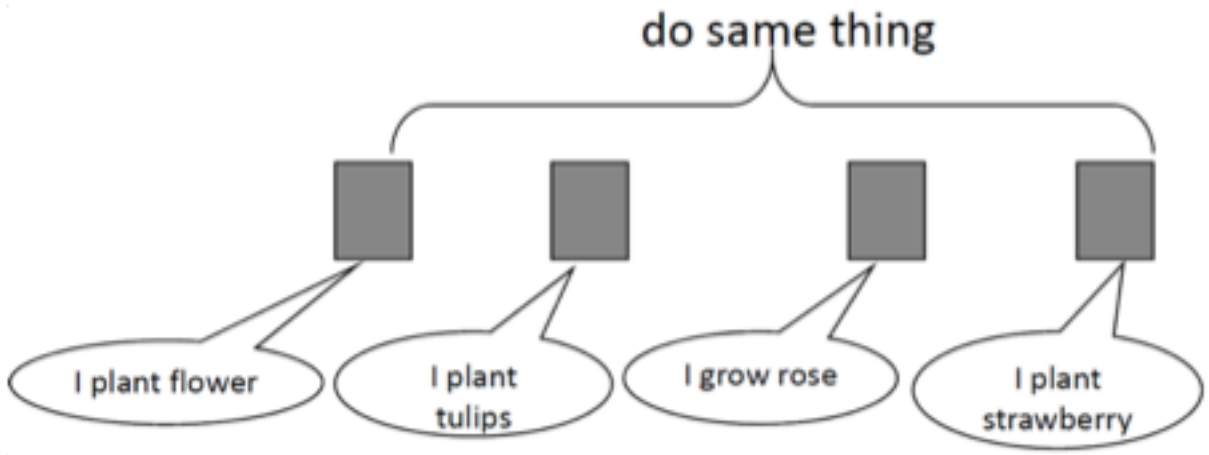
Design at the Routine Level: *Unacceptable* **Cohesion**

Logical Cohesion

Logical cohesion occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in.

The control flow or "logic" of the routine is the only thing that ties the operations together—they're all in a big if statement or case statement together.

```
public void sample( int flag ) {
   switch ( flag ) {
      case ON:
         // bunch of on stuff
         break;
      case OFF:
         // bunch of off stuff
         break;
      case CLOSE:
         // bunch of close stuff
         break;
      case COLOR:
         // bunch of color stuff
         break;
   }
}
```



Design at the Routine Level: *Unacceptable* **Cohesion**

Logical Cohesion

Logical cohesion occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in.

It's usually all right, to create a logically cohesive routine if its code consists solely of a series of if or case statements and calls to other routines.

if the routine's only function is to dispatch commands and it doesn't do any of the processing itself, that's usually a good design.

*The technical term for this kind of routine is "**event handler**"* An event handler is often used in interactive environments such as the Windows and Linux GUI environments.

Design at the Routine Level: *Unacceptable* **Cohesion**

Coincidental Cohesion

Coincidental cohesion occurs when the operations in a routine have no discernible relationship to each other

It's hard to convert coincidental cohesion to any better kind of cohesion—you usually need to do a deeper redesign and reimplementation

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

- Fix Car
- Bake Cake
- Walk Dog
- Fill our Astronaut-Application Form
- Get out of Bed
- Go the the Movies

Design at the Routine Level: *Bad* **Coupling**

Tight Coupling

Large dependence on the structure of one module by another.

Design at the Routine Level: *Good* **Coupling**

Loose Coupling

Modules with loose coupling are more independent and easier to maintain

Design at the Routine Level: *Worst* **Coupling**

Content Coupling

A module changes another module's data

Design at the Routine Level: *Not Worst* **Coupling**

Common Coupling

This occurs when all modules reference the same global data structure

Design at the Routine Level: *Not Worst* **Coupling**

External Coupling

Modules communicate through an external medium, such as files

Design at the Routine Level: *Acceptable* **Coupling**

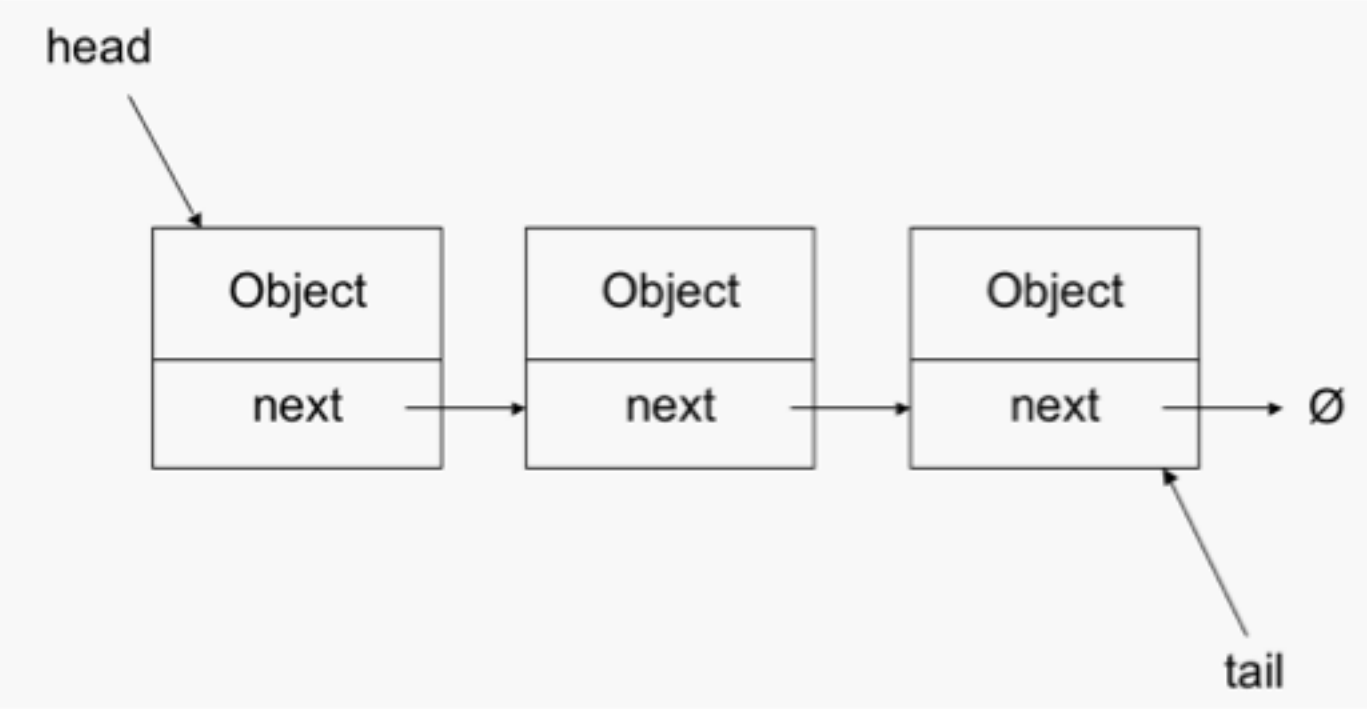
Control Coupling

Two modules exhibit control coupling if one (``module A'') passes to the other (``module B'') a piece of information that is intended to control the internal logic of the other.

Design at the Routine Level: *Acceptable* **Coupling**

Stamp Coupling

Two modules (``A'' and ``B'') exhibit stamp coupling if one passes directly to the other a ``composite'' piece of data-that is, a piece of data with meaningful internal structure -such as a record (or structure), array, or (pointer to) a list or tree.



Design at the Routine Level: *Ideal* **Coupling**

Modules *A* and *B* have the lowest possible level of coupling -no coupling at all -if they have no direct communication and are also not ``tied together'' by shared access to the same global data area or external device.

it implies that A and B be implemented, tested, and maintained (almost) completely independently; neither will affect the behavior of the other

Good Routine Names

A good name for a routine clearly describes everything the routine does

Describe everything the routine does

describe all the outputs and side effects. If a routine computes report totals and opens an output file, ComputeReportTotals() is not an adequate name for the routine. ComputeReportTotalsAndOpenOutputFile() is an adequate name but is too long and silly.

Avoid meaningless, vague, or wishy washy verbs

Some verbs are elastic, stretched to cover just about any meaning. Routine names like HandleCalculation() , PerformServices() , OutputUser() , ProcessInput() , and DealWithOutput() don't tell you what the routines do.

Don't differentiate routine names solely by number

Make names of routines as long as necessary

To name a function, use a description of the return value

describe all the outputs and side effects. If a routine computes report totals and opens an output file, ComputeReportTotals() is not an adequate name for the routine. ComputeReportTotalsAndOpenOutputFile() is an adequate name but is too long and silly.

To name a procedure, use a strong verb followed by an object

A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus object name.

Establish conventions for common operations

In some systems, it's important to distinguish among different kinds of operations. A naming convention is often the easiest and most reliable way of indicating these distinctions

Use opposites precisely

add/remove	increment/decrement	open/close
begin/end	insert/delete	show/hide
create/destroy	lock/unlock	source/target
first/last	min/max	start/stop
get/put	next/previous	up/down
get/set	old/new	

How Long Can a Routine Be?

The theoretical best maximum length is often described as one screen or one or two pages of program listing, approximately **50 to 150 lines**. In this spirit, IBM once limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976)

A large percentage of routines in object-oriented programs will be accessor routines, which will be very short. From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to 100–200 lines (A line is a non comment, nonblank line of source code).

How to Use Routine Parameters?

Interfaces between routines are some of the most error-prone areas of a program

One often-cited study by Basiliand Perricone (1984) found that **39 percent** of all errors were internal interface errors—errors in communication between routines.

How to Use Routine Parameters?

Put parameters in input-modify-output order

Instead of ordering parameters randomly or alphabetically, list the parameters that are input-only first, input-and-output second, and output-only third

Ada uses *in* and *out* keywords to make input and output parameters clear.

Ada Example of Parameters in Input-Modify-Output Order

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);
...

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);
...

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

How to Use Routine Parameters?

Consider creating your own in and out keywords

C++ Example of Defining Your Own *In* and *Out* Keywords

```
#define IN
#define OUT
void InvertMatrix(
   IN Matrix originalMatrix,
   OUT Matrix *resultMatrix
);
...

void ChangeSentenceCase(
   IN StringCase desiredCase,
   IN OUT Sentence *sentenceToEdit
);
...

void PrintPageNumber(
   IN int pageNumber,
   OUT StatusType &status
);
```

How to Use Routine Parameters?

If several routines use similar parameters, put the similar parameters in a consistent order

The order of routine parameters can be a mnemonic, and inconsistent order can make parameters hard to remember.

strncpy

<cstring>

```
char * strncpy ( char * destination, const char * source, size_t num );
```

Copy characters from string

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

destination and *source* shall not overlap (see memmove for a safer alternative when overlapping).

Parameters

`destination`
Pointer to the destination array where the content is to be copied.

`source`
C string to be copied.

`num`
Maximum number of characters to be copied from source.
size_t is an unsigned integral type.

memcpy

<cstring>

```
void * memcpy ( void * destination, const void * source, size_t num );
```

Copy block of memory

Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.

The underlying type of the objects pointed to by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed to by both the *destination* and *source* parameters, shall be at least *num* bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach).

Parameters

`destination`
Pointer to the destination array where the content is to be copied, type-casted to a pointer of type `void*`.

`source`
Pointer to the source of data to be copied, type-casted to a pointer of type `const void*`.

`num`
Number of bytes to copy.
size_t is an unsigned integral type.

How to Use Routine Parameters?

Use all the parameters

If you pass a parameter to a routine, use it. If you aren't using it, remove the parameter from the routine interface.

Unused parameters are correlated with an increased error rate. In one study, 46 percent of routines with no unused variables had no errors, and only 17 to 29 percent of routines with more than one unreferenced variable had no errors (Card, Church, and Agresti1986).

How to Use Routine Parameters?

Put status or error variables last

By convention, status variables and variables that indicate an error has occurred go last in the parameter list. They are incidental to the main purpose of the routine, and they are output-only parameters, so it’s a sensible convention.

How to Use Routine Parameters?

Don’t use routine parameters as working variables

It’s dangerous to use the parameters passed to a routine as working variables. Use local variables instead.

Java Example of Improper Use of Input Parameters

```
int Sample( int inputVal ) {
   inputVal = inputVal * CurrentMultiplier( inputVal );
   inputVal = inputVal + CurrentAdder( inputVal );
   ...
   return inputVal;
}
```

At this point, *inputVal* no longer contains the value that was input.

Java Example of Good Use of Input Parameters

```
int Sample( int inputVal ) {
   int workingVal = inputVal;
   workingVal = workingVal * CurrentMultiplier( workingVal );
   workingVal = workingVal + CurrentAdder( workingVal );
   ...

   ...
   return workingVal;
}
```

If you need to use the original value of *inputVal* here or somewhere else, it’s still available.

How to Use Routine Parameters?

Document interface assumptions about parameters

If you assume the data being passed to your routine has certain characteristics, document the assumptions as you make them. Even better than commenting your assumptions, use assertions to put them into code

- Whether parameters are input-only, modified, or output-only
- Units of numeric parameters (inches, feet, meters, and so on)
- Meanings of status codes and error values if enumerated types aren't used
- Ranges of expected values
- Specific values that should never appear

How to Use Routine Parameters?

Limit the number of a routine's parameters to about seven

Seven is a magic number for people's comprehension

If you find yourself consistently passing more than a few arguments, the **coupling among your routines is too tight**. Design the routine or group of routines to reduce the coupling. *If you are passing the same data to many different routines, group the routines into a class and treat the frequently used data as class data*.

How to Use Routine Parameters?

Make sure actual parameters match formal parameters

Formal parameters, also known as "dummy parameters," are the variables declared in a routine definition. Actual parameters are the variables, constants, or expressions used in the actual routine calls.

A common mistake is to put the wrong type of variable in a routine call

CHECKLIST: High-Quality Routines

Big-Picture Issues

- ❑ Is the reason for creating the routine sufficient?
- ❑ Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- ❑ Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- ❑ Does the routine's name describe everything the routine does?
- ❑ Have you established naming conventions for common operations?
- ❑ Does the routine have strong, functional cohesion—doing one and only one thing and doing it well?
- ❑ Do the routines have loose coupling—are the routine's connections to other routines small, intimate, visible, and flexible?
- ❑ Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

Parameter-Passing Issues

- ❑ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- ❑ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ❑ Are interface assumptions documented?
- ❑ Does the routine have seven or fewer parameters?
- ❑ Is each input parameter used?
- ❑ Is each output parameter used?
- ❑ Does the routine avoid using input parameters as working variables?
- ❑ If the routine is a function, does it return a valid value under all possible circumstances?

SUMMARY

The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.

Sometimes the operation that most benefits from being put into a routine of its own is a simple one.

You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.

The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.

Defensive Programming

The idea is based on *defensive driving*. In defensive driving, you adopt the mind-set that you're never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won't be hurt. You take responsibility for protecting yourself even when it might be the other driver's fault.

Defensive Programming

Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think.

Defensive Programming

```java
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:20)
```

Defensive Programming

```java
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nException: %s\n",
               inputMismatchException );
            scanner.nextLine(); // discard input so user can try again
            System.out.println(
               "You must enter integers. Please try again.\n" );
         } // end catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
         } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Protecting Your Program from Invalid Inputs

In school you might have heard the expression, "Garbage in, garbage out." That expression is essentially software development's version of caveat emptor: let the user beware.

For production software, garbage in, garbage out isn't good enough. A good program never puts out garbage, regardless of what it takes in.

- Check the values of all data from external sources
- Check the values of all routine input parameters
- Decide how to handle bad inputs

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

When an assertion is true, that means everything is operating as expected. When it's false, that means it has detected an unexpected error in the code.

Assertions are especially useful in large, complicated programs and in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

An assertion usually takes two arguments**:** a **Boolean expression** that describes the assumption that's supposed to be true, and a **message** to display if it isn't.

```
1  // Fig. 13.9: AssertTest.java
2  // Demonstrates the assert statement
3  import java.util.Scanner;
4
5  public class AssertTest
6  {
7     public static void main( String args[] )
8     {
9        Scanner input = new Scanner( System.in );
10
11       System.out.print( "Enter a number between 0 and 10: " );
12       int number = input.nextInt();
13
14       // assert that the absolute value is >= 0
15       assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17       System.out.printf( "You entered %d\n", number );
18    } // end main
19 } // end class AssertTest
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

You use assertions primarily for debugging and identifying logic errors in an application. They are comment-like code

You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program's user.

Users should not encounter any Assertion Errors through normal execution of a properly written program. Such errors should only indicate bugs in the implementation. E.g., **Debug mode vs. Release mode**

Assertions

An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs

- That an input parameter's value falls within its expected range (or an output parameter's value does)
- That a file or stream is open (or closed) when a routine begins executing (or when it ends executing)
- That a file or stream is at the beginning (or end) when a routine begins executing (or when it ends executing)
- That a file or stream is open for read-only, write-only, or both read and write
- That the value of an input-only variable is not changed by a routine
- That a pointer is non-null
- That an array or other container passed into a routine can contain at least X number of data elements
- That a table has been initialized to contain real values
- That a container is empty (or full) when a routine begins executing (or when it finishes)
- That the results from a highly optimized, complicated routine match the results from a slower but clearly written routine

Assertions: **Guidelines for Using Assertions**

Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur

Assertions check for conditions that should never occur. Error-handling code checks for off-nominal circumstances that might not occur very often, but that have been anticipated by the programmer who wrote the code and that need to be handled by the production code. Error handling typically checks for bad input data; assertions check for bugs in the code.

Assertions: **Guidelines for Using Assertions**

Avoid putting executable code into assertions

Putting code into an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions.

Visual Basic Example of a Dangerous Use of an Assertion

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```

Visual Basic Example of a Safe Use of an Assertion

```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

Assertions: **Guidelines for Using Assertions**

Do not use assertions for argument checking in public methods

Argument checking is typically part of the published specifications (or contract) of a method, and these specifications must be obeyed whether assertions are enabled or disabled

Erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException)

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 * rate > MAX_REFRESH_RATE.
*/
public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);
    setRefreshInterval(1000/rate);
  }
```

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate)
 *
 * @param  interval refresh interval in milliseconds.
*/
 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

Assertions: **Guidelines for Using Assertions**

Use Assertions for Internal Invariants

An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

Assertions: **Guidelines for Using Assertions**

Use Assertions for Internal Invariants

An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

Assumption: the suit variable will have one of only four values. To test this assumption, you should add the following default case:

```
default:
    assert false : suit;
```

```
switch(suit) {
  case Suit.CLUBS:
    ...
  break;

  case Suit.DIAMONDS:
    ...
  break;

  case Suit.HEARTS:
    ...
    break;

  case Suit.SPADES:
      ...
}
```

Assertions: **Guidelines for Using Assertions**

Use Assertions for Control Flow Invariants

place an assertion at any location you assume will not be reached

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    // Execution should never reach this point!!!
}

void foo() {
    for (...) {
      if (...)
        return;
    }
    assert false; // Execution should never reach this point!
}
```

Assertions: **Guidelines for Using Assertions**

Use assertions to document and verify preconditions and postconditions

Preconditions are the properties that the client code of a routine or class promises will be true before it calls the routine or instantiates the object. *Preconditions are the client code's obligations to the code it calls*.

Postconditions are the properties that the routine or class promises will be true when it concludes executing. *Postconditions are the routine's or class's obligations to the code that uses it*.

Visual Basic Example of Using Assertions to Document Preconditions and Postconditions

```
Private Function Velocity ( _
   ByVal latitude As Single, _
   ByVal longitude As Single, _
   ByVal elevation As Single _
   ) As Single

   ' Preconditions
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )

   ...

   ' Postconditions
   Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

   ' return value
   Velocity = returnVelocity
End Function
```

If the variables *latitude*, *longitude*, and *elevation* were coming from an external source, invalid values should be checked and handled by error-handling code rather than by assertions.

Assertions: **Guidelines for Using Assertions**

For highly robust code, assert and then handle the error anyway

Visual Basic Example of Using Assertions to Document Preconditions and Postconditions

```
Private Function Velocity ( _
   ByRef latitude As Single, _
   ByRef longitude As Single, _
   ByRef elevation As Single _
   ) As Single

   ' Preconditions
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )
   ...

   ' Sanitize input data. Values should be within the ranges asserted above,
   ' but if a value is not within its valid range, it will be changed to the
   ' closest legal value
   If ( latitude < -90 ) Then
      latitude = -90
   ElseIf ( latitude > 90 ) Then
      latitude = 90
   End If
   If ( longitude < 0 ) Then
      longitude = 0
   ElseIf ( longitude > 360 ) Then
   ...
```

Here is the assertion code.

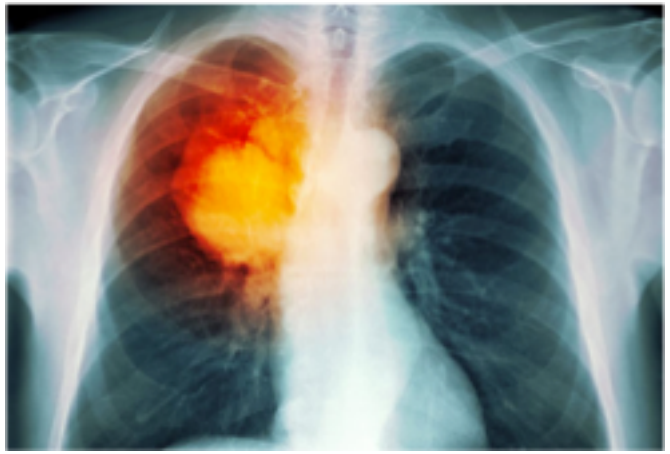
Here is the code that handles bad input data at run time.

Error-Handling Techniques

Return a neutral value

Sometimes the best response to bad data is to continue operating and simply return a value that's known to be harmless.

- A numeric computation might return 0.
- A string operation might return an empty string, or a pointer operation might return an empty pointer.
- A drawing routine that gets a bad input value for color in a video game might use the default background or foreground color.



Error-Handling Techniques

Substitute the next piece of valid data

When processing a stream of data, some circumstances call for simply returning the next valid data.

If you're reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record.

If you're taking readings from a thermometer 100 times per second and you don't get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

Error-Handling Techniques

Return the same answer as the previous time

Error-Handling Techniques

Substitute the closest legal value

In some cases, you might choose to return the closest legal value. This is often a reasonable approach when taking readings from a calibrated instrument

The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0, which is the closest legal value.

Cars use this approach to error handling whenever going back. Since a speedometer doesn't show negative speeds, when it simply shows a speed of 0—the closest legal value.

Error-Handling Techniques

Log a warning message to a file

When bad data is detected, you might choose to log a warning message to a file and then continue on.

This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data.

If you use a log, consider whether you can safely make it publicly available or whether you need to encrypt it or protect it some other way.

Error-Handling Techniques

Return an Error Code

You could decide that only certain parts of a system will handle errors. Other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.

- Set the value of a status variable
- Return status as the function's return value
- Throw an exception by using the language's built-in exception mechanism

Call an error-processing routine/object

Centralize error handling in a global error-handling routine or error-handling object.

Error-Handling Techniques

Display an error message wherever the error is encountered

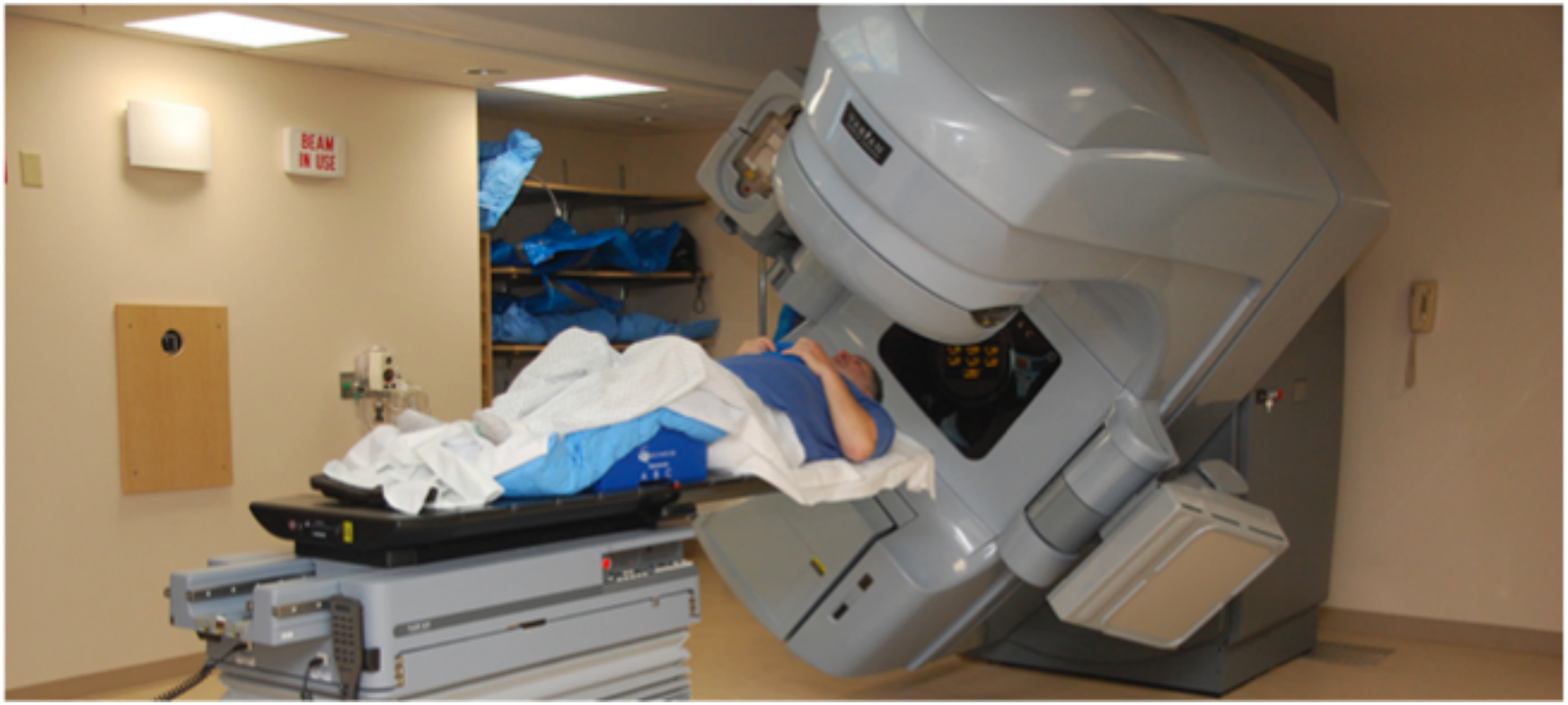
This approach minimizes error-handling overhead; however, it does have the potential to spread user interface messages through the entire application-how to separate UI. ***Tight coupling***

Beware of telling a potential attacker of the system too much. Attackers sometimes use error messages to discover how to attack a system.

Error-Handling Techniques

Shutdown

Some systems shut down whenever they detect an error. This approach is useful in safety-critical applications.



Error-Handling Techniques: **Correctness vs. Robustness**

Correctness means never returning an inaccurate result;
returning no result is better than returning an inaccurate result.

Robustness means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.

Safety-critical applications tend to favor correctness to robustness. It is better to return no result than to return a wrong result. e.g. the radiation machine

Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down.

Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

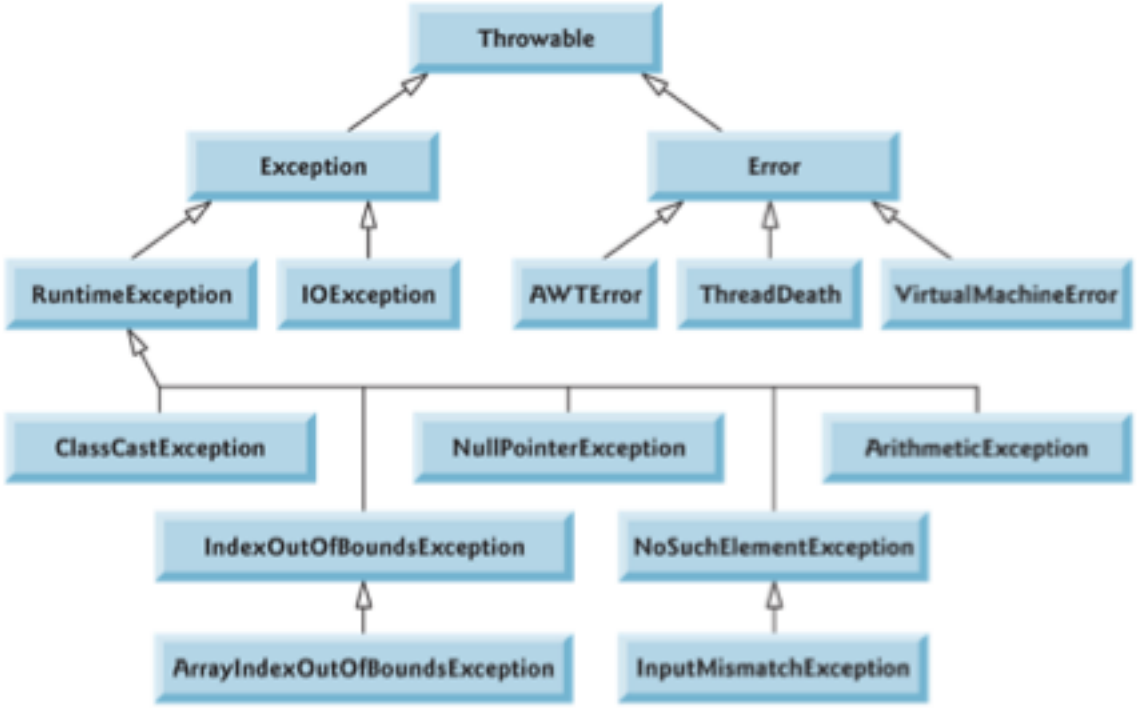
If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception, essentially throwing up its hands and yelling, "**I don't know what to do about this—I sure hope somebody else knows how to handle it!**"

Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Exception Attribute	C++	Java	Visual Basic
Try-catch support	yes	yes	yes
Try-catch-finally support	no	yes	yes
What can be thrown	*Exception* object or object derived from *Exception* class; object pointer; object reference; data type like string or int	*Exception* object or object derived from *Exception* class	*Exception* object or object derived from *Exception* class
Effect of uncaught exception	Invokes *std::unexpected()*, which by default invokes *std::terminate()*, which by default invokes *abort()*	Terminates thread of execution if exception is a "checked exception"; no effect if exception is a "runtime exception"	Terminates program
Exceptions thrown must be defined in class interface	No	Yes	No
Exceptions caught must be defined in class interface	No	Yes	No



Exceptions

Use exceptions to notify other parts of the program about errors that should not be ignored

The benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996)

Other approaches to handling errors create the possibility that an error condition can propagate through a code base undetected. Exceptions eliminate that possibility.

Exceptions

Throw an exception only for conditions that are truly exceptional

Exceptions should be reserved for conditions that are truly exceptional—in other words, for conditions that cannot be addressed by other coding practices

Exceptions represent a tradeoff between a powerful way to handle unexpected conditions on the one hand and increased complexity on the other.

Exceptions

Don’t use an exception to pass the buck

If an error condition can be handled locally, handle it locally. Don’t throw an uncaught exception in a section of code if you can handle the error locally.

Avoid throwing exceptions in constructors and destructors unless you catch them in the same place

The rules for how exceptions are processed become very complicated very quickly when exceptions are thrown in constructors and destructors.

Exceptions

Throw exceptions at the right level of abstraction

A routine should present a consistent abstraction in its interface, and so should a class. The exceptions thrown are part of the routine interface, just like specific data types are.

Here is the declaration of the exception that's at an inconsistent level of abstraction.

Bad Java Example of a Class that Throws an Exception at an Inconsistent Level of Abstraction

```
class Employee {
   ...
   public TaxId GetTaxId() throws EOFException {
      ...
   }
   ...
}
```

Here is the declaration of the exception that contributes to a consistent level of abstraction.

Good Java Example of a Class that Throws an Exception at a Consistent Level of Abstraction

```
class Employee {
   ...
   public TaxId GetTaxId() throws EmployeeDataNotAvailable {
      ...
   }
   ...
}
```

Exceptions

Include in the exception message all information that led to the exception

Be sure the message contains the information needed to understand why the exception was thrown.

If the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

Exceptions

Avoid empty catch blocks

Either the code within the try block is wrong because it raises an exception for no reason, or the code within the catch block is wrong because it doesn't handle a valid exception.

Bad Java Example of Ignoring an Exception

```
try {
   ...
   // lots of code
  ...
} catch ( AnException exception ) {
}
```

Good Java Example of Ignoring an Exception

```
try {
   ...
   // lots of code
  ...
} catch ( AnException exception ) {
   LogError( "Unexpected exception" );
}
```

Know the exceptions your library code throws

If you're working in a language that doesn't require a routine or class to define the exceptions it throws, be sure you know what exceptions are thrown by any library code you use.

Exceptions

Consider building a centralized exception reporter

Exceptions provide the means to separate the details of what to do when something out of the ordinary. Error detection, reporting, and handling often lead to confusing spaghetti code

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exceptions

Consider building a centralized exception reporter

Exceptions provide the means to separate the details of what to do when something out of the ordinary. Error detection, reporting, and handling often lead to confusing spaghetti code

Visual Basic Example of a Centralized Exception Reporter, Part 1

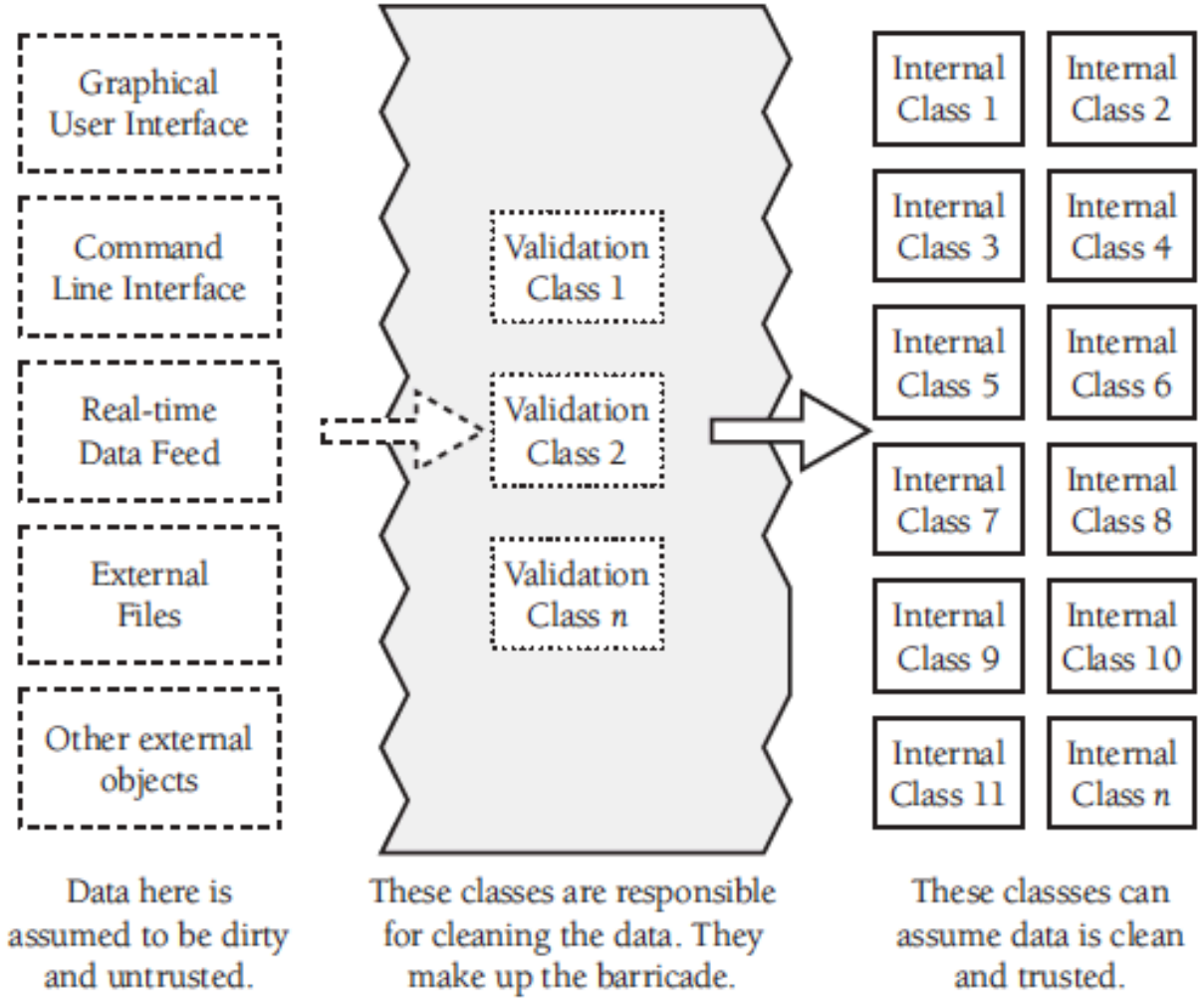
```
Sub ReportException( _
   ByVal className, _
   ByVal thisException As Exception _
)
   Dim message As String
   Dim caption As String

   message = "Exception: " & thisException.Message & "." & ControlChars.CrLf & _
      "Class:   " & className & ControlChars.CrLf & _
      "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
   caption = "Exception"
   MessageBox.Show( message, caption, MessageBoxButtons.OK, _
      MessageBoxIcon.Exclamation )

End Sub
```

Visual Basic Example of a Centralized Exception Reporter, Part 2

```
Try
   ...
Catch exceptionObject As Exception
   ReportException( CLASS_NAME, exceptionObject )
End Try
```

Barricade Your Program to Contain the Damage Caused by Errors

Barricades are a damage-containment strategy. The reason is similar to that for having isolated compartments in the hull of a ship.



Debugging Aids

Don’t Automatically Apply Production Constraints to the Development Version

A common programmer blind spot is the assumption that limitations of the production software apply to the development version

Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

Introduce Debugging Aids Early

The earlier you introduce debugging aids, the more they’ll help

Debugging Aids

Use Offensive Programming

Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running

- Make sure assert/abort the program. Don't allow programmers to get into the habit of just hitting the Enter key to bypass a known problem. Make the problem painful enough that it will be fixed.
- Completely fill any memory allocated so that you can detect memory allocation errors.
- Completely fill any files or streams allocated to flush out any file-format errors.
- Be sure the code in each case statement's default or else clause fails hard (aborts the program) or is otherwise impossible to overlook.
- Fill an object with junk data just before it's deleted.
- Set up the program to e-mail error log files to yourself so that you can see the kinds of errors that are occurring in the released software, if that's appropriate for the kind of software you're developing.

Debugging Aids

Plan to Remove Debugging Aids

If you're writing code for your own use, it might be fine to leave all the debugging code in the program.
If you're writing code for commercial use, the performance penalty in size and speed can be prohibitive.

- Use version-control tools and build tools like ant and make
- Use a built-in preprocessor
- Write your own preprocessor
- Use debugging stubs

Determining How Much Defensive Programming to Leave in Production Code

Leave in code that checks for important errors

Leave in code that helps the program crash gracefully

Log errors for your technical support personnel

Make sure that error messages you leave in are friendly

Being Defensive About Defensive Programming

Think about where you need to be defensive, and set your defensive programming priorities accordingly

CHECKLIST: Defensive Programming

General

- ❑ Does the routine protect itself from bad input data?
- ❑ Have you used assertions to document assumptions, including preconditions and postconditions?
- ❑ Have assertions been used only to document conditions that should never occur?
- ❑ Does the architecture or high-level design specify a specific set of error-handling techniques?
- ❑ Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- ❑ Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- ❑ Have debugging aids been used in the code?
- ❑ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ❑ Is the amount of defensive programming code appropriate—neither too much nor too little?
- ❑ Have you used offensive-programming techniques to make errors difficult to overlook during development?

Exceptions

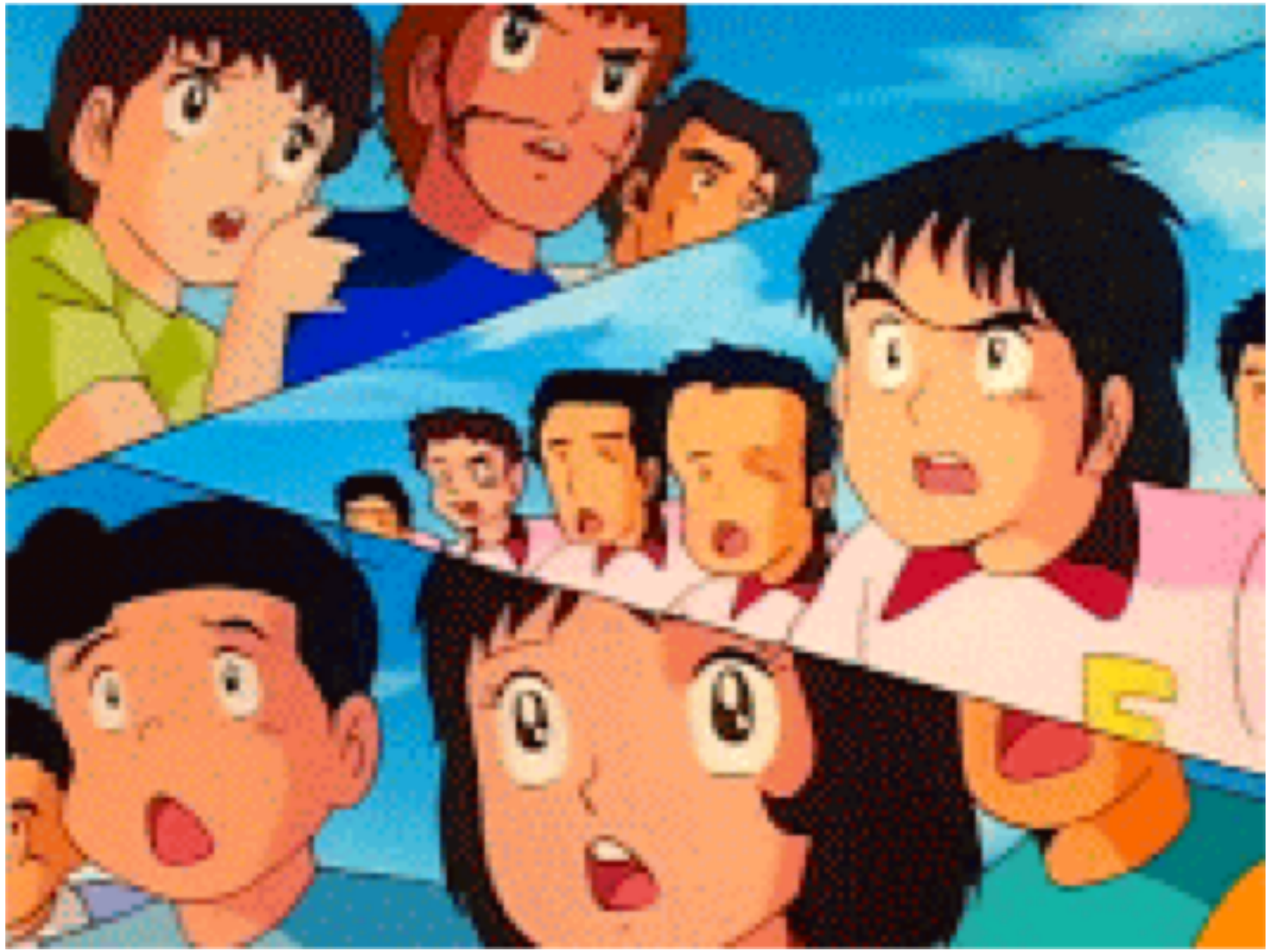
- ❑ Has your project defined a standardized approach to exception handling?
- ❑ Have you considered alternatives to using an exception?
- ❑ Is the error handled locally rather than throwing a nonlocal exception, if possible?
- ❑ Does the code avoid throwing exceptions in constructors and destructors?
- ❑ Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- ❑ Does each exception include all relevant exception background information?
- ❑ Is the code free of empty *catch* blocks? (Or if an empty *catch* block truly is appropriate, is it documented?)

Security Issues

- ❑ Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, HTML injection, integer overflows, and other malicious inputs?
- ❑ Are all error-return codes checked?
- ❑ Are all exceptions caught?
- ❑ Do error messages avoid providing information that would help an attacker break into the system?

SUMMARY

- Production code should handle errors in a more sophisticated way than “garbage in, garbage out.”
- Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.
- Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.
- The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.
- Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer’s intellectual toolbox when used with care, and they should be weighed against other error-processing techniques
- Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.



つづく