IT3205: Fundamentals of Software Engineering (Compulsory)

BIT – 2nd Year

Semester 3

IT3205: Fundamentals of Software Engineering

Coding

Duration: 3 hours

Learning Objectives

- Select appropriate programming language and development tools for a given problem
- Identify the features of a good program, good programming practices and program documentation

Detailed Syllabus

5.1 Programming languages and development tools

5.2 Selecting languages and tools

5.3 Good programming practices

What is Coding (Implementation)?

- Transforms the design specification to source code that can be executed on a computer.
- This is the final stage of the series of front end activities we have been dealing with.
- Coding is relatively straight forward given a design specification.
- Coding is a minor activity compared to the other phases of development.
- A good design may be spoiled by the bad choice of a language.
- However, a bad design cannot be corrected through coding.

What is Coding (Implementation)?

- Choice of the language and the coding style are important issues to consider.
- The programmer translates the design into source code of the chosen programming language.
- The language translator converts the source code in to executable code in several steps.
- Certain design issues may not be supported by the language in which case the coder may choose to violate design.
- Although design quality should not be compromised because of a language issue, design approach may depend on the language choice.

Coding

- The goal of the coding is to implement the design in the best possible manner.
- Coding activity affects both testing and maintenance phases.
- Time spent in coding is a small percentage of the total software cost, but the goal should be to reduce the cost of later phases.
- Having readability and understandability as a clear objective of the coding activity can itself help in producing software that is more maintainable.

5.1 PROGRAMMING LANGUAGES AND DEVELOPMENT TOOLS

Language Features

This table describes the strengths and weaknesses of some of the more important languages in current use.

Language	Features	Strengths	Weaknesses
Ada	- Procedural - Some object orientation built in exception handling. - Strong type checking - No pointers - Defined standards	- Strong type checking reduces errors on applications. - US DOS support - Good for safety critical and military applications. - Compliers meet the defined standards.	- Large run time system requires. - Expensive to develop applications. - Requires powerful machine to run on.
C	- Procedural - Weak type checking - Very low level pointers - Flexible	- Close to hardware/OS - Fast and efficient applications can be built. - Widely used	- Poor exception handling support. - Memory handling leads to unreliable code. - Compilers all different, don't meet standard.

Language Features

Language	Features	Strengths	Weaknesses
C++	- OO extension to C. - Weak type checking - Flexible - Pointers	- All those of C and OO concepts of Polymorphism, Inheritance (single & multiple), Encapsulation etc.	- As for C and can simple write C.
COBOL	- Procedural - Strong I/O handling for transaction processing (TP). - Defined standards	- Suited to batch TP applications - Widely used - Most 4GLs interface to COBOL.	- Large run time system required. - Old - many features simply added on later. - Not all compilers meet standard.
Fortran	- Procedural - Strong arithmetic support through libraries.	- Suited to data analysis where significant arithmetic processing required.	- Old - more modern languages provide most of the features.

Language Features

Language	Features	Strengths	Weaknesses
Java	- Object Oriented - Better type checking than C but still reasonably weak. - Standard defined by Sun Microsystems.	- Platform independent. - Dynamic downloading of classes. - Good user interface and network support through libraries. - Ideal for network applications.	- Requires own run time environment. - Controlled by a commercial organization.
Pascal	- Procedural - Strong type checking - Defined standard	Good teaching language	- Not widely used in industry - Poor environment support
Visual Basic	- Simple procedural language. - Interpreted - Extensive Windows programming support	- Suited to small applications and prototyping. - Some OO concepts in user interface handling. - Widely used	- Performance - Complex data structures cannot be modeled.

Language Features

Language	Features	Strengths	Weaknesses
Visual C++	- C++ programming environment for MS Windows	- Support for windows programming. - User interface design. - Syntax sensitive editors. - Some code generation	- Portability of code.
C #	- Object oriented language derived from C++ and Java. - .NET includes a Common Execution engine and a rich class library. - The classes and data types are common to all of the .NET languages	- In C# Microsoft has taken care of C++ problems such as Memory management, pointers. - It supports garbage collection, automatic memory management and a lot.	- Cannot perform unsafe casts like convert double to a Boolean.

5.2 SELECTING LANGUAGES AND TOOLS

Selecting Languages and Tools

- There's no language suitable for all tasks, and there probably won't ever be one.
- When choosing a programming language, you have to balance programmer productivity, maintainability, efficiency, portability, tool support, and software and hardware interfaces.
- Often, one of these factors will shape your decision. In other cases, the choice depends on the productivity you gain from certain language features, such as modularity and type checking, or external factors, such as integrated development environment support.
- Finally, for some tasks, adopting an existing domain-specific language, building a new one, or using a general-purpose declarative language can be the right choice.

Coding Practices

- Allow code to be written in a more predictable and maintainable fashion.
- While working code can be written without these techniques it is rare for such code to be maintainable other than by the original author.
- There are three main areas to consider;
  - **Reliability** ->Is the software fault free?

    Two complementary approaches:

    1. Fault avoidance-develop so that errors are not introduced.
    2. Fault tolerance-develop so that the program continues when faults occur.
  - **Readability** ->can be code to be easily maintained.
  - **Reuse** ->can the code be used again.

Programming for reliability

- Reliable code is that which conforms to its specification and continues in operation in all but most extreme circumstances.
- There are two techniques used to increase the reliability of code,
  1. avoidance and
  2. tolerance

Programming for reliability

- Following structured programming techniques leads to code that is less likely to have faults and is easier to correct.
  - Fault avoidance
    - Aims to ensure the code has few errors as possible.
  - Fault tolerance
    - Aims to produce code that will continue to function in the presence of errors.
    - Fault tolerance includes:
      - Exception Handling
      - Defensive programming
      - Fault recovery

Exception Handling

- An exception is an error or unexpected event. Traditional languages, Pascal, C etc. have no specific support for handling exceptions, newer languages, Ada, Java, have some in built exception handling.

Example:

- Procedure A calls Procedure B
- Procedure B calls Procedure C
- An exception occurs in procedure C
- B cannot continue
- Need to signal exception to A

Exception Handling

- With traditional languages we must resort to setting error or status variables which can be shared or passed from procedure to procedure,

  for example:

```
/* allocate an integer array of SIZE elements */
    int *intArray = (int *)malloc(SIZE);
    if (intArray =» NULL)
        {
        /* no heap memory available */
        error - MEMORY_ERROR;
        return -1;
        }
```

Still have to *unwind* the nested function calls.

Exception Handling

With Java exception handling is built into the language

Throw exception

```
class counter {
    int total;
    int number_values;
    // Counter methods
    public int Average() {
     if (this.number_values == 0)
       throw new DivideException (),
     else
     // division code
     }
    }
```

Catch exception

```
class myClass {
 // Class definition
 public void myMethod {
  counter counterOne;
  // Method implementation
  // Get the average
  try {
    counterOne.Average() ;
  } catch ( DivideException e) {
   System.out.println("Division by zero");
  }
 }
}
```

Defensive programming

- Assume that there are faults and inconsistencies in the system and validate data.
- Example checks are:
  - internal data states, for example probabilities between 0 and 1, money = integer + 2 decimal places etc.
  - checksums
  - array bounds checking
  - division by 0
  - pointer validation - check that pointer is allocated and points to a data structure.
  - If any problems occur then the system must recover to a safe state.

Fault recovery

- Forward recovery, correct damaged system state
- Error detection and correction of coded data
- File or database recovery
- Backward recovery, restore system to a safe state
- Database transaction roll-back

Programming for readability

- One of the major problems with reviewing and maintaining code is that the code is unreadable.
- A number of straight forward techniques are available for improving the readability of the code and therefore reducing review and maintenance effort.
  - Naming conventions
  - Data types
  - Control constructs

Naming conventions

- It is important on projects of more than one person or on the development of software applications which are to be maintained (all of them!) that some form of naming conventions are drawn up at the start of the project.
- The naming conventions would cover such things as program names, function and procedure names, variable names, constant names source file names and so on.
- This is important in reducing the effort required for anyone except the author to read and understand the code.

Naming conventions

- This is true during the development phase, when reviews and such like are carried out, and during the maintenance phase when the code has to be changed by someone unfamiliar with it.
- Typically large organizations will have company wide naming standards which must be followed by all projects.
- As well as following the standards laid down, names should always be meaningful, for example the name of a function should reflect the purpose of the function.

Naming conventions

- Example naming conventions for the 'C' language might be:
  1. All constant names are in upper case and begin C, for example C_SPEED_OF_LIGHT
  2. All variable names are in mixed case with each word beginning with a capital, variable names may be prefixed with a lower case letter indicating the type of the variable, i for int, l for long etc.,

     for example, `int iCurrentTime;`
  3. All variable names should reflect the use of the variable, i.e. it should describe the real world object represented.

     for example;

     ```
     List UnknownWordList; // Good
     List u_list; // Bad
     ```

     Loop indexes can use `i, j` etc.

Naming conventions

- Example naming conventions for the 'C' language might be:

4. Function and procedure names are in mixed case with each word beginning with a capital, function names may be prefixed with a lower case letter indicating the return type of the function, i for int, v for void etc.,

   for example,

```
void vCalculateTotal(List SubTotalList) {
}
```

5. Function and procedure names must describe the purpose of the function or procedure.
6. Source code file names must reflect the contents of the file

   for example,

   `CalculateTotal.c` - contains the function CalculateTotal

Data types

- Use abstract data types to make the code clearer.

Example:

```
Type TrafficLightColour is (red, amber, green);
ColourShowing, NextColour: TrafficLightColour;
```

Control constructs

- Use the standard flow control constructs for structured programming, sequence, selection and iteration, flow should be from the top of the program down.
- Loops, decision statements, routines etc. should all have single entry and exit points, avoid `gotos, breaks,exits` etc.
- Functions should have a single return.

Control constructs

- Keep code simple - one possible complexity heuristic is;

1. Start with 1 for the straight path through the routine
2. Add 1 for each control keyword, if, while, repeat, and, or etc.
3. Add 1 for each keyword in a case. If the case does not have a default it should.

– given the above total then judge the complexity as follows:
  - 1-5 Routine probably ok
  - 6-10 May need some re-design and/or simplification
  - >10 Too complex, redesign required.

– In addition to the above the amount of data, number of lines etc. could be taken into account.

Technical considerations

- **Environment**
  - the language should support the features of the environment in which the application is to run, for example if the application is to run on the Windows operating system then the programming language and development tools should help to reduce the effort required to produce the user interface, that is use *Visual C++ or Borland C++* rather than a basic C compiler and the Windows SDK.
- **Language support**
  - the language selected must have good support within the environment on which development will take place and in which the application will run. It is much easier to develop mainframe data processing applications in *COBOL* or some 4GL rather than 'C' or *Ada*.

Technical considerations

- **Performance**
  - applications which have critical speed and size performance requirements cannot usually be built using an interpreted language such as *Basic* or a language which requires a complex run time system such as *Smalltalk* or *Java* as the overhead in the run time system is too great.
- **Safety / security**
  - safety critical and secure applications usually require a programming language which supports these requirements. Thus a nuclear power station control system might be written in *Ada* rather than 'C'.
- **Interfaces to other systems**
  - if the application must interface to other systems then these may restrict the languages which can be used.

Non-technical considerations

Here we must look at the project as a whole and the business objectives of the organization. It should be noted that these considerations are typically more important than the purely technical factors.

- **Timescales**
  - the project timescales may restrict the choice of languages, for example selecting a language known by the programmers will reduce timescales, or help to meet timescales.
- **Maintenance**
  - if the application is to be passed on to a maintenance team then the skills of the maintenance team and the nature of the other applications being maintained may have a bearing on the language chosen. Do you want to have to re-train the maintenance staff?

Non-technical considerations

- **Other applications**
  - if all other applications built and maintained by the organization are written in COBOL, for example, then there is a very strong argument for building new applications in COBOL.
- **Available tools**
  - what compilers, debuggers are currently available in the workplace? Buying new software to support the development may increase the development costs significantly.

Non-technical considerations

- **Risk**
  - adopting new technology always carries an associated risk, on a project which is critical to the business such risks may be unacceptable.
- **Morale**
  - programmers love to use the latest technology, on a non-critical project it may be beneficial to use an innovative approach to development to retain the interest of the staff, for example Java could be used to develop some small in-house utility. This has the added advantage of bringing new technical knowledge into the organization.

5.3 GOOD PROGRAMMING PRACTICES

Good Programming Practices

Before you write one line of code, be sure you:

- Understand the problem you're trying to solve
- Understand basic design principles and concepts
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate
- Select a programming environment that provides tools that will make your work easier
- Create a set of unit tests that will be applied once the component you code is completed

Good Programming Practices

As you begin writing code, be sure you:

- Constrain your algorithms by following structured programming language
- Select data structures that will meet the needs of the design
- Understand the software architecture and create interfaces that are consistent with it
- Keep conditional logic as simple as possible
- Create nested loops in a way that makes them easily testable
- Select meaningful variable names and follow other local coding standards
- Write code that is self-documenting
- Create visual layout that aids understanding

Code Reviews

- A code review is also called technical review.
- The code review for a module is carried out after the module is successfully compiled and all the syntax errors eliminated.
- These are extremely cost effective strategies for reduction in coding error in order to produce high quality code.
- In a review, a work product is examined for defects by individuals other than the person who produced it.

Code Reviews

- A work product is any important deliverable created during the requirements, design, coding, or testing phase of software development.
  - Examples of work products are phase plans, requirements models, requirements and design specifications, user interface prototypes, source code, architectural models, user documentation, and test scripts.
- Reviews can be conducted by individuals or group
- There are two types of code reviews;
  1. Code walk-throughs
  2. Code inspection.

Code Reviews

- Code reviews can often find and remove common vulnerabilities such as format string exploits, race conditions, memory leaks and buffer overflows, thereby improving software security.
- Improve the quality of the code being reviewed
- Improve programmers

Code Reviews

- A code review is not a meeting to;
  - negatively criticize someone else's code
  - criticize architecture or design (unless it is an architectural or design review)
  - criticize a colleague
  - determine whether someone is to be removed from a project or not
  - determine whether someone is performing up to standard

Code Walkthrough

- Objectives of Code Walkthrough are;
  - To discover algorithmic and logical errors in the code.
  - To consider alternative implementations
  - To ensure compliance to standards & specifications

Code Walkthrough

- This an informal code analysis technique. In this technique when a module has been coded, it is compiled and eliminate all syntax errors.
- Some members of the development team are given the code few days before the walkthrough meeting to read and understand the code.
- Each member selects some test cases and simulates the execution of the code by hand.
- The team performing the code walkthrough consists of 3-7 members

Code Inspection

- Software maintenance is the general process of changing a system after it is diverted.
- This change includes corrections of coding errors, design errors, specification errors or accommodation of new requirements.
- There are three types of software maintenance:
  1. repair software faults
  2. adapt the software to a different operating system
  3. add or modify the system's functionality

Code Inspection

- Errors detected during code inspection;
  - Use of uninitialized variables
  - Jumps in to loops
  - Non terminating loops
  - Incompatible assignments
  - Array indices out of bounds
  - Improper storage allocation and deallocation
  - Mismatches between actual and formal parameters in procedure calls
  - Improper modification of loops