BCT 2308: SOFTWARE DEVELOPMENT TOOLS & ENVIRONMENTS

CHAPTER 4:

Role of Programming Languages in the Development Environments

Learning Outcomes

- By the end of this chapter, the learner should be able to:
  - Define a programming language.
  - Describe roles played by programming languages in software development environments.
  - Choose appropriate programming language for a software development project.

Introduction

- Programming languages are the programmer's primary means of expression.
- They have a long history, and a number of distinctive programming language styles have emerged.

Introduction

- Software tools offer a rich set of possibilities, starting with the tools most directly related to programming languages: compilers, a familiar figure from previous chapters, but also their sister tools, interpreters.

What is a Programming Language?

- A programming language is a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.

Introduction

- The term *programming language* usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal.
- Each programming language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Introduction

- High-level programming languages, while simple compared to human languages, are more complex than the languages the computer actually understands, called *machine languages*.
- Each different type of CPU has its own unique machine language.

Introduction

- Although implementation of a system or algorithm in a programming language is not the most critical phase in software design, it certainly is the one that makes software executable.

Introduction

- Thus, programming language play a central role in the development environment.
- They contribute to program documentation and modifiability and can make programs checkable statically, thus contributing to their reliability.

Introduction

- Furthermore, most modern programming languages are much more than simple coding tools.
- They increasingly invade the realm of traditionally recognized as design by providing constructs that help in structuring complex systems.

Introduction

- Consequently, the features they offer may either support or bias, or even resist good design techniques.

Introduction

- With respect to classification, programming languages are:
  - A formal tool.
  - A life cycle specific tool.
  - Monolingual tool, by definition.
  - A development tool.

Formal Tool

- Programming language syntax and semantics are defined vigorously and this provides the basis for their execution.

A life cycle specific tool

- They are typically devoted to the implementation phase, although some languages are used at least partially in design and even in the specification phase of the life cycle.

Monolingual tool, by definition

- The capability of including modules written in different languages in the executable code may be supported by the available polylingual environment.
  - A development tool.

Roles of Programming Languages

1. Modularity.
2. Handling of Anomalies.
3. Concurrency.
4. verification
5. Software Design.

Modularity

- Almost all practical programming languages provide features to decompose large programs into smaller components.

Modularity

- Even when the language does not provide specific modularization constructs, usually the implementation offers ad hoc features that enable software developers to put together separately developed parts to form a more complex system.

Modularity of C Programs

- A C program consists of a number of data and function definitions.
- One of the functions must be named main.
- main is the function that is invoked at the start of program execution.
- The closest notion to a module in C is a program file.

Modularity of C Programs

- Any variables or functions declared in one file are visible only in the file, but they may be imported explicitly by other files.
- E.g. a software engineer might have a file that contains all the common data types and variables.
- This file can be included by all other files in the program, or selectively by any file that needs it.

Modularity of C Programs

- A file that contains only declarations is called a header file.
- By convention, the names of such files ends with ".h" while the name of a normal C file ends in ".c".

Modularity of C Programs

- The same modularization approach is used to integrate the use of libraries into the language.
- Each library is known both by the object code for the routines it supports and by a header file that contains the declarations for the supported routines.

Modularity of C Programs

- E.g. the C standard input-output library has two components i.e. the object code to be linked into programs and a header file called stdio.h that contains the declarations for all the functions that the library supports e.g. printf and scanf.

Modularity of C Programs

- Any program that wants to use the library must include the corresponding header file so that the compiler can check for the validity of the calls in the program.
- A C module may easily be used to build an abstract object.

Modularity of C Programs

- The C language supports program modularization based on files and a set of conventions.
- However, the language does not enforce the conventions.

Handling of Anomalies

- Anomalous situations may occur during execution, even if much care was applied in the design of a program.
- Anomalies must be handled carefully, possibly by applying separation of concerns and incrementally, e.g. one can first deal with normal execution, and then consider exceptional cases.

Handling of Anomalies

- However, it is still quite controversial whether, and if so, how techniques to cope with exceptions (exception-handling features) should be embedded in a programming language.

Handling of Anomalies

- Generally, the main objections to embedding exception-handling features in programming languages have to do both with methodology (what is a clean design structure for exception handling?) and with semantic definition and efficient implementation of the programming language.

Handling of Anomalies

- The exception-handling mechanism of a programming language provides the following capabilities:
  - Declaration of exceptions
  - Raising of exceptions
  - Handling of, and recovery from, exceptions.

Handling of Anomalies

- Declaration of exceptions:
  - Programming languages provide facilities to declare a possible anomalous situation and give it an identifier.
  - Moreover a set of built-in exceptions exists, such as Division_by_0 and Memory_Overflow.

Handling of Anomalies

- Raising of Exceptions:
  - This is used to signal the actual occurrence of the anomalous condition.
  - Such an occurrence is implicitly signaled in the case of built-in exceptions; e.g. Division_by_0 is automatically signaled by the hardware.
  - User defined exceptions must be raised explicitly, by executing the statement RAISE <exception identifier>.

Handling of and Recovery from Anomalies

- Once an exception has been signaled, an appropriate action must be executed in response to it.
- Quite possibly, the action may result in a recovery from the exception, say, by locally putting the system in a consistent state in such a way that normal execution may be resumed.
- In this case, the client does not even realize that an exception has occurred.

Handling of and Recovery from Anomalies

- However, if locally-initiated recovery is impossible, the exception may be propagated to another module, perhaps a higher-level module in some abstraction hierarchy, notifying that module of the failure of the current operation, so that appropriate recovery may be taken at the higher level.
- Or, the higher-level module may, in turn, propagate the exception further up the chain of modules.

Handling of and Recovery from Anomalies

- The choice of propagation strategy is one of the important semantic issues that faces the language designer in defining exception handling.
- Usually, propagation follows the currently active USES chain: control is first transferred to the client, then to the client's client, and so on.

Handling of and Recovery from Anomalies

- In the worst case, the whole program aborts because of an unrecoverable exception that was propagated all the way up to the main control routine.

Concurrency

- Traditionally, concurrency is not supported by the programming language, rather, it is provided by the operating system or even by directly accessing the underlying machine through interrupts.

Concurrency

- However, language-supported concurrent programming would have the following benefits:
  - Support for construction of traditional concurrent systems such as control systems and operating systems.
  - Speed up even many sequential algorithms

Concurrency: Speeding up Algorithms

- This can be done by exploiting the facilities for parallel computing that are now supplied by many hardware architectures.
- A low level feature provided by a few languages is the *coroutine* construct of Simula 67.
- This construct introduces a kind of "pseudoconcurrency" on a conventional sequential machine.

Concurrency: Speeding up Algorithms

- Co-routines run one at a time, so that there are no concurrently executed units.
- However, one can simulate concurrency by interleaving the execution of co-routines.
- Most modern languages that support concurrency use a process as a basic unit of concurrent execution.

Concurrency: Speeding up Algorithms

- A process may be executed asynchronously with respect to other processes.
- The decision whether logically concurrency is simulated by sharing a unique physical processor according to some policy or whether physical parallelism is achieved by allocating different processes to different processors is a matter left to the implementation.

Concurrency: Speeding up Algorithms

- In some languages such as concurrent Pascal, processes synchronize with each other through passive objects (monitors).
- In other languages, cooperation among processes is regulated through active processes that act as guardians of resources.

Verification

- Generally, good design techniques and languages that support them can favour the production of correct programs by supporting powerful static checks.

Verification

- They can also improve verifiability by providing simple control structures and suitable modularization primitives that make reasoning about the behaviour of programs simple, local and incremental.

Verification

- A few languages allow the programmer to add assertions to programs along the lines.
- This facility allows a specification language (an assertion language) to be integrated with the programming language.

Assertions

- Assertions are facts about the state of the program variables
- It is wasteful to spend your time looking at variables that are not effected by a particular statement
- Default assertion
  - any variable not mentioned in the assertion for a statement do not affect the state of computation

Use of Assertions

- Pre-condition
  - assertion describing the state of computation before statement is executed
- Post condition
  - assertion describing the state of computation after a statement is executed
- Careful use of assertions as program comments can help control side effects

Verification

- If tools are available to deal with assertions, the correct behaviour of the programs can then be assessed with respect to the stated predicates.
- A predicate is an expression that evaluates to TRUE, FALSE, or UNKNOWN.

Verification

- There are two ways to use assertions:
  - Proving the correctness of the program with respect to the assertions and
  - Monitoring the assertions at run time to detect whether they are violated.

Verification

```
assert(0 <= index && index < length);
```

Verification

- In proving the correctness of the program, a tool must be available to verify the program's correctness.
- The tool would be interactive and would support verification along the lines.

Verification

- In monitoring, assertions are turned into predicates that are checked at run time.

Software Design

- An appropriate programming language can support or even enforce good design techniques.
- However, one can write bad programs using a good language, as well as a good program using a poor language though with much difficulty.

Software Design

- Thus, the unavailability of the "ideal" programming language is not a valid excuse for a poor design structure or a sloppy programming style.

Software Design

- The following are examples of disciplined programming practices that may be applied to languages that do not support them directly:
  - If one uses a block structured language.
  - In FORTRAN or C, one may insert comments in subprogram headers to document the underlying USES relation.
  - Program assertions.
  - Even in an unstructured, low-level language.

Design using block structured languages

- This language does not provide external modules, one can still develop outer level procedures as if they were separate modules, although they cannot be compiled separately.

Design using block structured languages

- If such procedures access global variables, one may complement their headers with comments that explicitly state which global variables are read and/or written by the procedure.

Design using block structured languages

- Also, one can still design information-hiding modules even though they are not physically visible as such through language-defined constructs and even though the language cannot enforce protection of hidden secrets.

Design using block structured languages

- E.g. in order to implement an abstract object, one may use comments to bracket a set of procedures that represent access operations and use a comment to denote the hidden internal data structure.
- A similar solution may be adopted to document abstract data type.

Software Design: Assertions

- Program assertions can be quite useful, even when they are just treated as comments in the program.
- They can help a human reader in analyzing the program and they can be manually turned into test cases.

Software Design: Unstructured Languages

- Even in an unstructured, low-level language, one may use comments and indentation to illustrate the conventions that were followed in structuring a program, since these conventions might not be clear from the code.

Software Design: Unstructured Languages

- E.g. comments may be used to document an assembly language loop, or they may specify what the type of an operand should be in an untyped language.

Task

- Using sample code, explain how Visual Basic and Java implements the following:
  - Assertions.
  - Exception Handling

End

Next Chapter Ideal Scenario