BCT 2308:
SOFTWARE DEVELOPMENT TOOLS AND ENVIRONMENTS

CHAPTER 3:

Representative Tools

Chapter Objectives

- By the end of this chapter, the learner should be able to:
  - Describe some of the most commonly used tools and environments.
  - Classify tools and environments.

Introduction

- Software development is generally supported by tools, ranging from tools supporting a single activity to integrated environments supporting a complete development process.

Representative tools

- The list of commonly used tools is necessarily nonexhaustive, as new tools and new types of tools are continuously introduced.
- The list is a way of software developers to talk about some important categories of tools.

Representative Tools

- Editors
- Linkers
- Interpreters
- Code generators
- Debuggers
- Tools used in testing
- Static analyzers
- User-interface management tools
- Project management tools
- Configuration management tools

Editors

- Editors are a fundamental software development tool since software is ultimately a more or less complex collection of documents such as requirements specification, module architecture, and programs.

Editors

- Editors can be categorized in the following categories:
  - Textual or graphical.
  - Formal or informal.
  - Monolingual or polylingual.
- Editors are used to produce, correct or update documents.

Editors

- Editors can either be textual or graphical.
- Editors such as notepad are textual.
- Editors such as FrontPage HTML editors and graphical.

Editors

- Editors can follow a formal syntax, or they can be used for writing informal text or drawing free-form pictures. E.g. a general purpose graphical editor can be used to produce any kind of document, diagrams such as data flow diagrams.

Editors

- Similarly, a word processor may be used to write programs in any programming language, but the tool cannot help in finding missing keywords, ill formed expressions, and undeclared variables.

Editors

- In order to perform such checks, tools that are sensitive to the syntax and semantics of the language should be used.

Editors

- Editors can be either monolingual (syntax directed editor) or polylingual (generic syntax directed editor).
- A conventional word processor is intrinsically polylingual.
- Emacs edit is capable of operating in different modes.

Editors

- A mode may be driven by the syntax table of a particular language, allowing simple checks and standard indentation to be performed by the editor.

Editors

- Editors may be used not only to produce, but also to correct or update, documents.
- Thus editors should be flexible (e.g. able to run interactively or in batch) and easy to integrate with other tools.

Linkers

- These are tools that are used to combine mutually referencing object-code fragments into a larger system.
- They can both be monolingual (when they are language specific) or polylingual (when they can accept modules written in different languages).

Linkers

- Linkers basically bind names appearing in a module to external names exported by other modules.
- In case of language specific linkers, this may imply some kind of intermodule type checking, depending on the nature of the language.

Linkers

- A polylingual linker may perform only binding resolution, leaving all language specific activities to other tools.
- The concept of a linker has broader applicability than just to programming languages.

Linkers

- Typically, if one is dealing with a modular specification language, a linker for that language would be able to perform checking and binding across various specification modules.

Interpreters

- An interpreter executes actions specified in a formal notation (its input code).
- At one extreme, an interpreter is the hardware that executes the machine code.
- It is also possible to interpreter specifications, if they are written in a formal language.

Interpreters

- In this case, an interpreter behaves as a simulator or a prototype of the end product and can help detect mistakes in the specifications even in the early stages of the software development process.

Interpreters

- Requirements specification often occurs incrementally, hand in hand with the analysis of the application domain.
- In such cases, it is useful to check the requirements by suitable execution of an incompletely specified system.

Interpreters

- E.g. initially, one may decide to specify only the sequence of screen panels through which the end user will interact with the system, leaving out the exact specification of the functions that will be invoked in response to the user input.

Interpreters

- This decision might be dictated by the fact that, in the application under development, the user interfaces are the most critical factor affecting the requirements.
- Thus they can be checked with the end user whether the interaction style that is to be provided corresponds to his /her expectations.

Interpreters

- This implies that the interpreter of the specifications should be able to generate screen panels and should allow us to display sequences of screen panels in order to demonstrate the interactive sessions with the application.

Interpreters

- The interpreter should tolerate the incompleteness of specifications e.g. when no functions are provided in response to the various commands that might be entered in the fields on the screen panels.

Interpreters

- The interpreter operates like a partial prototype, allowing experimentation with the look and feel of the end product.
- In other cases, the result of interpreting the requirements is more properly called requirements animation.

Interpreters

- What is provided on the screen is a view of the dynamic evolution of the model, which corresponds to the physical behaviour of the specified system.
- For example, a finite state machine that is used to model the evolution of a state-changing dynamic system can be easily animated.

Interpreters

- If the state changing machine system is a plant controlled by a computer, and a finite state machine displayed on the terminal describes the states entered by the plant as a consequence of commands issued by the computer.

Interpreters

- Animation may be achieved by blinking the states of the finite state machine as the corresponding state of the plant is entered.
- For example, the control signal may be simulated by pressing any key on the terminal.

Interpreters

- Usually, the interpreters operate on actual input values.
- However, it is possible to design symbolic interpreters, which may operate on symbolic input values.
- Symbolic execution corresponds to a whole class of executions on input data.

Interpreters

- Thus, a symbolic interpreter can be a useful verification tool and can be used as an intermediate step in the derivation of test data that cause execution to follow certain paths.

Code Generators

- The software construction process is a sequence of steps that transform a given problem description (specification) into another description (implementation).
- Generally, the implementation description is more formal, more detailed, and lower level than the specification. Implementation is also more efficiently executable.

Code Generators

- The transformation process eventually ends in machine-executable code.
- Even intermediate steps may be executable.
- The additional transformation steps are passed through because interpreters of intermediate steps are generally slower than interpreter of the final product

Code Generators

- Derivation steps may require creativity and may be supported by tools to varying degrees.
- A simple and fully automatic step is the translation from source code into object code.
- This is performed by the compiler.

Code Generators

- Other derivation steps are the decomposition of modules according to hierarchy, can only be partially supported.
- An optimizer tools is essentially a translator supporting the stepwise transformations of specifications into an implementation.

Code Generators

- Moving from formal specifications of a module to an implementation may also be viewed as a transformation that involves creative activities such as designing data structures or algorithms.
- The clerical parts can be supported by automatic tools e.g. converting specifications into partial source code, leaving part of the creative part of the transformation to the designer.

Code Generators

- Examples of generalized code generators are provided by fourth generation tools which automatically generate third generation object code from a higher level languages (4^{th} generation languages).
- Such systems are centered around database systems.
- Screen panels for human-computer interaction may be generated for inserting or manipulating data in the database and for querying the database.

Code Generators

- Also, reports may be automatically generated from the database definition.
- The user can choose from several options to define the report formats.
- E.g. in a database of employees, a report on all employees that match a selection criteria can automatically be generated by specifying simple declarative options, with no need for defining and coding report generation algorithms.

Debuggers

- Debuggers may be viewed as a kind of interpreter.
- They execute a program with the purpose of helping to localize and fix errors, by applying the principles of the programming language.

Debuggers

- Modern debuggers give the user capabilities to:
  1. Inspect the execution state in a symbolic way. Symbolic means it uses symbolic identifiers of program objects, not that the debugger is a symbolic interpreter.
  2. Select the execution mode such as initiating step-by-step execution or setting breakpoints symbolically.

Debuggers

3. Select the portion of the execution state and the execution points to inspect without manually modifying the source code. This makes debugging simpler and also avoids the risk of introducing foreign code that one may forget to remove after debugging.
4. Check intermediate assertions.

Assertion

- Assertion is a statement in the program code that enables a programmer to test his/her assumptions about the program.
- The syntax is *assert expression*;

Debuggers

- Apart from just locating and removing defects from a program, a good symbolic debugger can be used to observe the dynamic behaviour of programs.
- Symbolic debugger is able to understand words and symbols.

Debuggers

- It can accelerate the validation of behavioral specifications by allowing a user to interact with an executing code at the source level.
- In response to a user query, the debugger must retrieve and display the value of a source variable in a manner consistent with user expectations with respect to the source statement where execution has halted.

Debuggers

- By animating the program in this way, a debugger can be a useful aid in understanding programs written by another programmer, thus supporting program modification and reengineering.

Tools used for software Testing

- Testing may be supported by tools in several ways.
- The main categories of tools used for testing are:
  - Tools supporting documentation of testing
  - Tools supporting test-case derivation
  - Tools supporting an evaluation of the testing activity.
  - Tools supporting testing of other software qualities.

Tools supporting documentation of testing

- These tools support the bookkeeping of test cases, by providing forms for test case definition, storage, and retrieval.
- A typical form that can be used for describing a test case is shown on the next slide.

Tools for Documenting Testing

Project Name:	Date of Test:
Tested function:	
Tested Module:	
Test Case Description:	
Description of Results:	
Comments:	

Tools supporting test-case derivation

- A test case is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Tools supporting test-case derivation

- A test case is a set of conditions or variables under which a tester will determine if a requirement upon an application is partially or fully satisfied.
- It may take many test cases to determine that a requirement is fully satisfied.

Tools supporting test-case derivation

- In order to fully test that all the requirements of an application are met, there must be at least one test case for each requirement unless a requirement has sub requirements.
- In that situation, each sub requirement must have at least one test case.

Tools supporting test-case derivation

- Techniques for building test cases can be made effective by supporting tools even though they cannot provide completely mechanical solutions.
- The main prerequisite for tools that generate test cases is the availability of a formal description.

Tools supporting test-case derivation

- Thus white-box techniques can be naturally implemented in an automatic or semiautomatic way, since they apply to programs, which are formal descriptions of algorithms.

Tools supporting test-case derivation

- To apply white-box techniques, the first step is to use a symbolic interpreter to derive path conditions and then synthesize a set of test values that satisfy those conditions and guarantee traversal of the desired paths.
- In the case of black box testing, test cases can be derived automatically only if formal specifications are available.

Tools for evaluating testing

- These include tools that provide various kinds of metrics, such as the number of statements executed, the percentage of paths covered in the control flow graph, and reliability and software science measures.

Tools for evaluating testing

- E.g. a software developer may decide that testing is sufficiently complete when a certain reliability level is reached according to some known reliability model.

Tools for testing of other software qualities

- Testing may be used to evaluate qualities such as performance.
- Performance monitors help in keeping track of, and verify execution time, main memory usage, and other performance related parameters.

Static Analyzers

- Program analysis can be considered to be a verification technique that complements testing.
- Similarly, dynamic verification tools such as interpreters and debuggers, and other testing supporting tools, can be complemented by tools that are devoted to certifying some desired properties without executing the program.

Static Analyzers

- The features offered by such tools depend highly on the type of property under consideration.
- E.g. one might desire a deadlock detection tool to help in designing deadlock-free concurrent systems, or a tool for timing analysis to prove that a critical real-time system always responds to a certain input stimuli within a specified time constraint.

Static Analyzers

- Several important properties of programs can be classified as either flow properties, i.e. properties are of the variation of some relevant entities, particularly data and control during program execution.
- Thus, tools for flow analysis are static tools that characterize certain dynamic properties of the flow of data and control.

Static Analyzers

- Data and flow analyzers can help in the discovery of errors.
- E.g. data flow analysis can reveal the use of uninitialized variables and control flow analysis can determine whether there are any unreachable statements in the program.

Static Analyzers

- However, such properties are undecidable in the general case.
- Thus, the associated tools may fail to give absolutely precise answers.

Static Analyzers

- E.g. a data flow analyzer could report some variable as potentially uninitialized, even if no execution of the program will ever reference the variable before initializing it first.
- An interesting use of data flow analysis is building specialized analyzers for particular applications.

Static Analyzers

- E.g. in some secure applications, one should prove that no data are transferred from certain modules to other modules.

Static Analyzers

- In this connection, one may think of a tool that allows the user to define a particular flow property of interest and automatically generates an analyzer that checks for the presence of that property.
- Flow analyzers can also support program transformation, whether automatic or not. E.g. many techniques for code optimization are based on data flow analysis.

User-interface Management Tools

- Evolution of technology has greatly improved the friendliness of human-computer interaction.
- This is particularly relevant when the end user has little or no technical background, but it is also appreciated by software engineers.

User-interface Management Tools

- Therefore, the design of good user interfaces has increasingly become an integral part of the development of software products.

User-interface Management Tools

- Most modern user interfaces are built from a set of common concepts that have become fairly standard. Some of these concepts are:
  - Windows
  - Icons
  - Dialog boxes
  - Diagrams

User-interface Management Tools

- The spread of standard user interface built from a common set of abstractions has encouraged the development of tools that support the definition and implementation of interfaces for different applications.
- E.g. some support tools for 4TH Generation languages provide primitives such as define_dialog_box, define_menu.

User-interface Management Tools

- The definition of a dialog box is accomplished quite naturally in two phases.
  - First, its logical contents are defined, i.e. the field and types of the items represented in the box.
  - Then its graphical layout is defined.

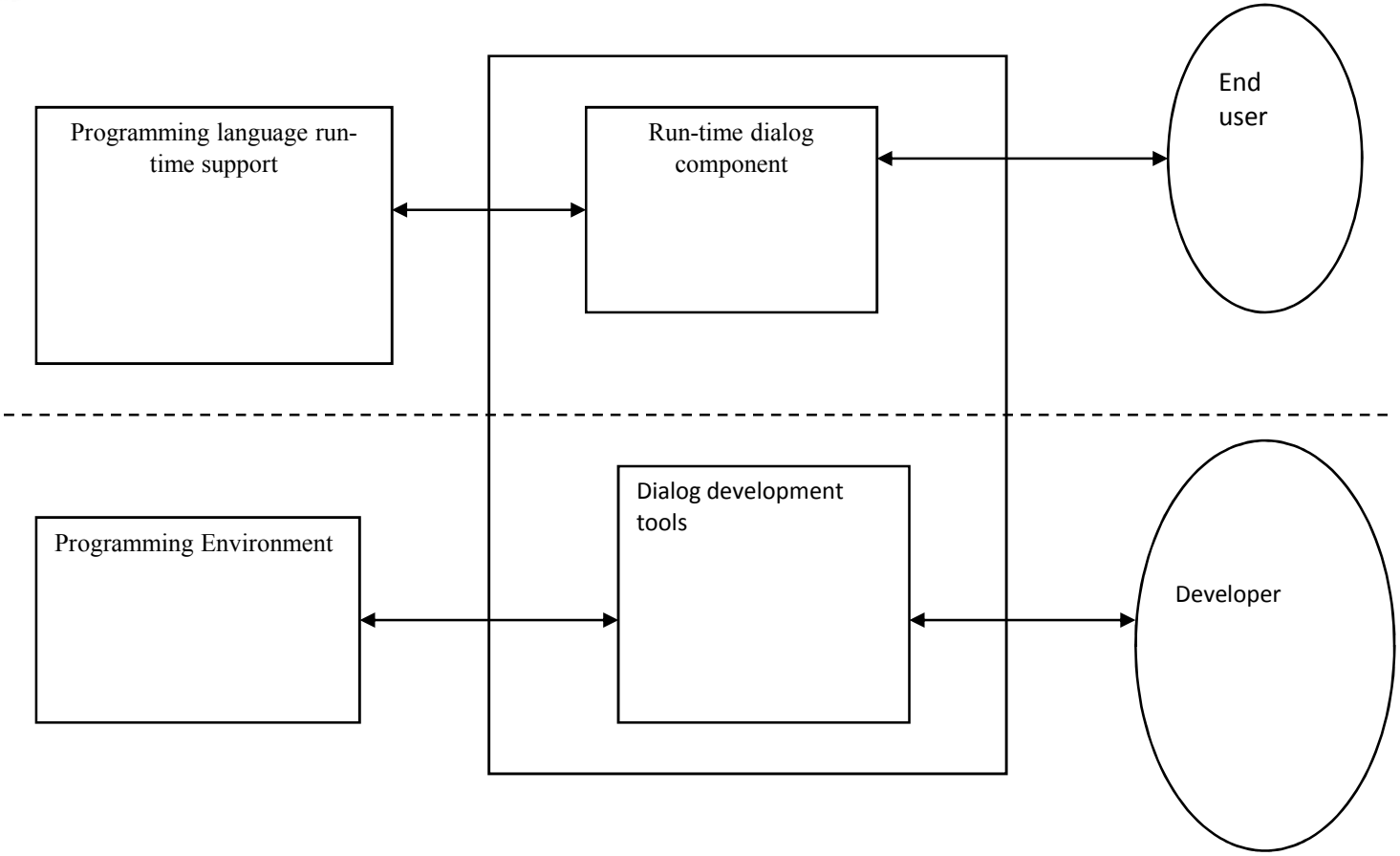
User-interface Management Tools

- A *User interface management Systems* (UIMS) provides a set of basic abstractions (icons, menus, etc) that may be used to customize a variety of interfaces.
- It also provides a library of run-time routines to be linked to the developed application in order to support input and output.

User-interface Management Tools

- Thus, a UIMS falls both under the category of development tools and under the category of end-product components

Two-component UIMS



User-interface Management Tools

- The run-time component of the UIMS manages the bindings between internal data structures of an application and the external views of the application.

User-interface Management Tools

- The internal structure of the data is the one that is manipulated by the application, while the external structure is the one that is visualized on the screen for human-computer interaction.

Configuration Management Tools

- Software development involves many activities and the production of many types of artifacts.
- Configuration management deals with coordinating activities of different individuals and teams involved in software production.
- Furthermore, since software evolves over time, most of these documents too evolve.

Configuration Management Tools

- And many of the documents have to exist in several versions.
- E.g. after performing some code optimization, the original program is kept until the new version has been verified to be working, or different versions of the manual, one fore each type of machine that supports the application.

Configuration Management Tools

- Therefore configuration management also deals with controlling the change and evolution of software artifacts.

Configuration Management Tools

- Configuration management tools should provide the following functions:

  1. The capability of defining, storing, and retrieving variants and revisions of a program.

Configuration Management Tools

2. The capability of controlling access to libraries of components e.g. preventing two members of a software team from checking out the same component for modification at the same time.
3. The capability of keeping a system in a consistent state by automatically generating derived versions upon modification of some components.

Configuration Management Tools

- These functions are integrated together with the services offered by the software engineering database.
- Examples of configuration management tools supported within UNIX environments are:
  - Source Code Control System (SCCS).
  - Make.

Project Management Tools

- Management issues grow more critical as the complexity of a project increases.
- Thus the usefulness of management tools becomes more relevant in large, multi-person projects.
- Most management principles and techniques are quite general and are not specific to software.

Project Management Tools

- However, a few of principles such as cost estimation techniques are heavily dependent on software.
- Management activities may be supported by a variety of tools, some devoted to management in general and others specific to the software development projects.

Project Management Tools

- General management support facilities include scheduling and control tools based on PERT diagrams and Gantt charts.
- Features typically offered by such tools are the capability of interacting both textually and graphically.

Project Management Tools

- E.g. the duration of an activity and its precedence relations with other activities may initially be given textually, and then may be visualized graphically.
- Microsoft Project is an example of project management tools.

Project Management Tools

- The tool may:
  - Calculate the starting and ending dates for activities.
  - Balance resource loading
  - Verify that a resource is not assigned beyond its maximum availability.

Project Management Tools

- Among software related facilities are the cost estimation tools based on the estimation techniques such as COCOMO model and function-point based models.
- In the case of cost estimation, some tools allow the equation underlying the estimation method to be adapted to the specific organization, depending on historical data collected in that organization.

Project Management Tools

- Tools supporting source-code metrics can also provide useful information to managers.

Software Engineering Infrastructures

- Integrated project Support environment is an integrated collection of tools supporting all phases of the software production process.
- This requires that artifacts produced in the process to be collected in a repository to be accessed by different tools and saved for later use.

Software Engineering Infrastructures

- Such repository, together with the operations needed to manage it, constitutes the heart of the environment.
- The services provided by the infrastructure correspond to what is called software engineering database.

Software Engineering Infrastructures

- This database includes such facilities as data or document retrieval, updating, and composition.
- It should support:
  1. The management of baselines, including access control to the stored components by different individuals.

Software Engineering Infrastructures

2. "Browsing" through repository, following the many relations that exist among different items. E.g. one could look for all modules that are used by a given module, for one or more implementations of a given interface of a given module, for all programmers that have been working on some subsystem, or for cost estimates related to the development of a library.

Software Engineering Infrastructures

3. Deductive capability to exploit nontrivial logical links between different items. E.g. given a library of modules, each described by a formal interface specification and its implementations.

Software Engineering Infrastructures

- Other services provided by the infrastructure may be viewed as functions at the level of the operating system.
- They are:
  1. Support for the concurrent execution of several related activities, possibly through a window system.

Software Engineering Infrastructures

2. Support for the distribution of data and activities on different nodes of a distributed architecture. A typical architecture of an environment is in fact a set of workstations connected through a local network. Each node is the personal workstations of a software engineer, and the infrastructure ensures the cooperation of individual nodes. The system must also provide mechanisms to support distributed transactions on the network.

The end of Chapter 3

Practice: Assertions in Java and Visual Basic