Implementation
and Maintenance

Implementation and Maintenance

- Implementation
  - Coding
  - Unit/Component testing
  - Integration and System testing
  - Release/Acceptance testing
- Postdelivery maintenance
  - Corrective maintenance
  - Perfective maintenance
  - Adaptive maintenance
- Retirement

2

Coding

- Choice of programming language
- Good programming practice
- Coding standards
- Code reuse

3

Choice of Programming Language

- The language is usually specified in the contract
- But what if the contract specifies that :
  - The product is to be implemented in the "most suitable" programming language
- How to choose a programming language
  - Cost–benefit analysis
  - Compute costs and benefits of all relevant languages

4

Languages

- First generation languages
  - Machine languages (binary machine code instructions)
- Second generation languages
  - Assemblers (symbolic notation such as "mov $4, next")
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++, Java)
  - One 3GL statement is equivalent to 5–10 assembler statements
- Fourth generation languages (4GLs)
  - Non-procedural language (easier to code) : Natural?
  - Each 4GL statement was intended to be equivalent to 30 or even 50 assembler statements

5

Good Programming Practice

- Use of *consistent* and *meaningful* variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers

6

Use of Consistent and Meaningful Variable Names

- A code artifact includes the variable names :
  - `freqAverage, frequencyMaximum,`
  - `minFr, frqncyTotl`
- What is the problem ?
  - A maintenance programmer has to know if `freq, frequency, fr, frqncy` all refer to the same thing
- What do instead :
  - If so, use the identical word, preferably `frequency,` perhaps `freq` or `frqncy`,but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity

7

Use of Consistent and Meaningful Variable Names

- We can use `frequencyAverage, frequencyMyaximum, frequencyMinimum, frequencyTotal`
- We can also use `averageFrequency, maximumFrequency, minimumFrequency, totalFrequency`
- But all four names must come from the same set

8

Self-Documenting Code

- Self-documenting code is exceedingly rare
- The key issue: Can the code artifact be understood easily and unambiguously by
  - The SQA team
  - Maintenance programmers
  - All others who have to read the code

9

Self-Documenting Code

- Example:
  - Code artifact contains the variable : `xCoordinateOfPositionOfRobotArm`
  - This is abbreviated to `xCoord`
- Is that fine to abbreviate?
  - This is fine, because the entire module deals with the movement of the robot arm
- What about for other users of the code ?
  - But does the maintenance programmer know this?

10

Prologue Comments

The name of the code artifact
A brief description of what the code artifact does
The programmer's name
The date the code artifact was coded
The date the code artifact was approved
The name of the person who approved the code artifact
The arguments of the code artifact
A list of the name of each variable of the code artifact, preferably in alphabetical order, and a brief description of its use
The names of any files accessed by this code artifact
The names of any files changed by this code artifact
Input–output, if any
Error-handling capabilities
The name of the file containing test data (to be used later for regression testing)
A list of each modification made to the code artifact, the date the modification was made, and who approved the modification
Any known faults

11

Other Comments

- Suggested that
  - Comments are essential whenever the code is written in a non-obvious way
  - or when makes use of some subtle aspect of the language
- Often a bad sign !
- Instead :
  - Recode in a clearer way
  - We must never promote/excuse poor programming
  - However, comments can assist future maintenance programmers

12

Use of Parameters

- When is it justifiable to have "constants" in your code ?
- There are almost no genuine constants
- One solution:
  - Use **const** statements (C++), or
  - Use **public static final** statements (Java)
- A better solution:
  - Read the values of "constants" from parameter file

13

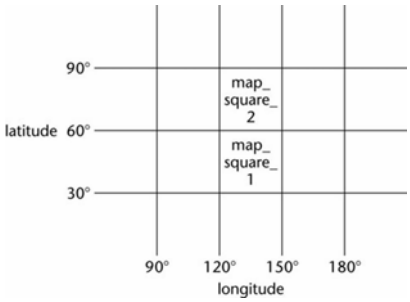
Code Layout for Increased Readability

- Use indentation
- Better, use a pretty-printer
- Use plenty of blank lines
  - To break up big blocks of code

14

Example: Nested if statements

- Problem
  - A map consists of two squares. Write code to determine whether a point on the Earth's surface lies in map square 1 or map square 2, or is not on the map



15

Example: Nested if statements

- Solution 1. Badly formatted

```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```

16

Example: Nested if statements

- Solution 2. Well-formatted, badly constructed

```
if (latitude > 30 && longitude > 120)
{
  if (latitude <= 60 && longitude <= 150)
    mapSquareNo = 1;
  else
    if (latitude <= 90 && longitude <= 150)
      mapSquareNo = 2;
    else
      print "Not on the map";
}
else
  print "Not on the map";
```

17

Example: Nested if statements

- Solution 3. Acceptably nested

```
if (longitude > 120 && longitude <= 150 && latitude > 30 && latitude <= 60)
  mapSquareNo = 1;
else
  if (longitude > 120 && longitude <= 150 && latitude > 60 && latitude <= 90)
    mapSquareNo = 2;
  else
    print "Not on the map";
```

18

Example: Nested if statements

- A combination of `if-if` and `if-else-if` statements is usually difficult to read
- Simplify: The `if-if` combination

```
if <condition1>
        if <condition2>
```

  is frequently equivalent to the single condition

```
if <condition1> && <condition2>
```

- Rule of thumb :
  - `if` statements nested to a depth of greater than three should be avoided as poor programming practice

19

Programming Standards

- Standards can be both a blessing and a curse
- Modules of coincidental cohesion arise from rules like
  - "Every module will consist of between 35 and 50 executable statements"
- Better
  - "Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements"

20

Remarks on Programming Standards

- No standard can ever be universally applicable
- Standards imposed from above will be ignored
- Standard must be checkable by machine
- The aim of standards is to make maintenance easier:
  - If they make development difficult, then they must be modified
  - Overly restrictive standards are counterproductive
  - The quality of software suffers

21

Examples of Good Programming Standards

- "Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the team leader"
- "Modules should consist of between 35 and 50 statements, except with prior approval from the team leader"
- "Use of `gotos` should be avoided. However, with prior approval from the team leader, a forward `goto` may be used for error handling"

22

Code Reuse

- Code reuse is the most common form of reuse
  - Subroutine libraries, e.g., multiplication or I/O routines, etc.
  - Runtime support routines
  - Large-scale statistical libraries (SPSS), or numerical analysis libraries
  - Class libraries
  - ...
- Note that artifacts from all workflows can be reused

23

Integration

- When ?
  - The general approach:
    - Implementation followed by integration
  - Better:
    - Combine implementation, testing and integration methodically

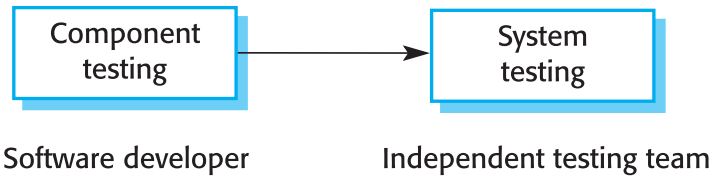
24

Testing

- Component testing
  - Testing of individual program components;
  - Usually the responsibility of the component developer (except sometimes for critical systems);
  - Tests are derived from the developer's experience.
- System testing
  - Testing of groups of components integrated to create a system or sub-system;
  - The responsibility of an independent testing team;
  - Tests are based on a system specification.

25

Testing



26

Goals

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements;
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification;
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

27

Component Testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Components may be:
  - Individual functions or methods within an object;
  - Object classes with several attributes and methods;
  - Composite components with defined interfaces used to access their functionality.

28

System Testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
  - Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
  - Release testing - the test team test the complete system to be delivered as a black-box.

29

Integration Testing

- Involves building a system from its components and testing it for problems that arise from component interactions.
- Top-down integration
  - Develop the skeleton of the system and populate it with components.
- Bottom-up integration
  - Integrate infrastructure components then add functional components.
- To simplify error localisation, systems should be incrementally integrated.

30

Release Testing

- Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system
  - Choose inputs that force the system to generate all error messages;
  - Design inputs that cause buffers to overflow;
  - Repeat the same input or input series several times;
  - Force invalid outputs to be generated;
  - Force computation results to be too large or too small.

31

Maintenance

- Classical maintenance
  - Development → Installation → Maintenance model
  - Classification as development or maintenance depends on the time at which an activity is performed

32

Maintenance

- In 1995, the International Standards Organization (ISO) and International Electrotechnical Commission (IEC) defined maintenance *operationally*
- Maintenance is nowadays defined as:
  - The process that occurs when a software artifact is modified because of a problem or because of a need for improvement or adaptation
  - Regardless of whether this takes place before or after installation of the software product
- ISO/IEC definition adopted by IEEE and EIA

33

Maintenance

- *Postdelivery maintenance*
  - Changes after delivery and installation [IEEE 1990]
- *Modern maintenance* (or just *maintenance*)
  - Corrective, perfective, or adaptive maintenance performed at any time [ISO/IEC 1995, IEEE/EIA 1998]

34

Postdelivery Maintenance

- Is it a bad sign that your software undergoes maintenance?
  - No ! Bad software is discarded !
- Good software is maintained, for 10, 20 years or more
  - How long does development take?
- Why there is always maintenance?
  - Find extended uses of existing software
  - Software is a model of reality, which is constantly changing

35

Corrective Maintenance

- Concerned with fixing reported failures (errors) observed in the software
  - Coding errors
    - typically easy and inexpensive to correct since they are confined within one unit
  - Design errors
    - more expensive since they may involve changes to several units
  - Requirements errors
    - most expensive since they often involve extensive system redesign (re architecting) to correct

36

Adaptive Maintenance

- Involves changing the software to work in some new environment such as a different machine or operating system
  - Characterized by no change in functionality, just a move to the new environment

37

Perfective Maintenance

- Involves implementing new or changed functionality due to changes in requirements
  - Normally generated either by users (customers) of the software,
    e.g., need to handle a new transaction or a new kind of bank card or service
  - Or by changes in the business environment the software operates in,
    e.g., changes to tax laws, new information interchange formats, competition from MS, etc.

38

Now, in your project ...

- Final bundle, containing
  - ***Source Package***
    - **Source files**
      - Source files should be provided in a well organized group of directories.
      - Sources of extra tools, utilities, extensions are provided in a separate directory.
    - **Developers guide**
      - For anyone to make additions and maintenance on your software, guidelines giving the internals, file formats, design structure, the identification of the source components.
    - **Compilation manual**
      - Compilation dependencies: Which compilers, tools, libraries, environment required for compilation. Be specific about the versions.
      - Compilation process: Step by step explanation of how the sources can be compiled.

39

Now, in your project ...

  - ***Binary Package***
    - **Binary files**
      - All necessary binaries of your software should be provided in a common binary package format. Self installable formats like installation executables, file formats like rpm, and deb are preferred.
    - **Installation manual**
      - Install dependencies: O.S., environment, run-time library, other software dependencies required to install and use your software. Be specific about the versions.
      - Guideline: step by step descriptions to install your software.
    - **Administration and User manual**
      - How to use the programs. Configuration files, how to configure the programs. Execution parameters if any. Troubleshooting.

40

Now, in your project ...

- Please, do not forget to include the creation commands for your database tables.

41