CS342 Software Engineering

Dr. Ismail Hababeh
German Jordanian University

Lecture 17

IMPLEMENTATION WORKFLOW

Adapted from Software Engineering, by Dr. Paul E. Young
& slides by Dr. Mohammad Daoud

Overview

- The implementation workflow
- Choose of programming language
- Good programming practice
- Self-Documenting Code
- Programming Comments
- Coding standards
- Testing the implementation

The Implementation Workflow

- The aim of the implementation workflow is to implement the target software product in the selected implementation language
- A large software product is partitioned into subsystems
- The subsystems consist of *components* or *code artifacts*

Programming Team

- Real-life products are generally too large to be implemented by a single programmer
- Therefore, programming team work on implementing the software product.

The Implementation Languages

- First generation languages
  - Machine languages
- Second generation languages
  - Assemblers
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++, Java)

Fourth generation Languages 4GLs

- Fourth generation languages (4GLs)
  - Each 4GL statement was intended to be equivalent to 30 or even 50 assembler statements
  - There are literally hundreds of 4GLs
- 4gl examples: PowerBuilder, Visual DataFlex, LabVIEW, Oracle

4GLs Advantages

- Speed up application-building (increase productivity)
- Result in applications that are
  - Easy to build
  - Quick to change
- Reduce maintenance costs
- Simplify debugging
- User friendly
  - Leading to end-user programming
- Many 4GLs are supported by powerful CASE environments which identifies sets of services.

4GLs Field Oriented

- Market share
  - No one of the 4GL dominates the software market, because each of 4GLs hasn’t all the necessary features
- Conclusion
  - Selecting the appropriate 4GL with care.

Fifth Generation Languages 5GLs

- Based on problem-solving.
- Using constraints given to the program.
- Used mainly in artificial intelligence AI research.
- Designed to make the computer smarter.
- Examples of 5GLs:
  - Mercury: pure logic programming language designed for creation of large, fast, reliable programs.
  - OPS5: rule-based or production system computer language
  - Prolog: used for natural language understanding and expert systems

Good Programming Practice

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

Use of Consistent Variable Names

- For Example, a code artifact includes the variable names `freqAverage`, `frequencyMaximum`, `minFr`, `frqncyTotl`
- A maintenance programmer has to know if `freq`, `frequency`, `fr`, `frqncy` all refer to the same thing, this may cause misunderstanding.
- Solution: use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`

  Or, use a different word (e.g., `rate`) for a different quantity

Use of Meaningful Variable Names

- Use of prefix-meaning variables:

  `frequencyAverage, frequencyMaximum,`

  `frequencyMinimum, frequencyTotal`

- Use of postfix-meaning variables:

  `averageFrequency, maximumFrequency,`

  `minimumFrequency, totalFrequency`

- All variable names must come from the same set

Use of Constants

- 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 a parameter file

Self-Documenting Code

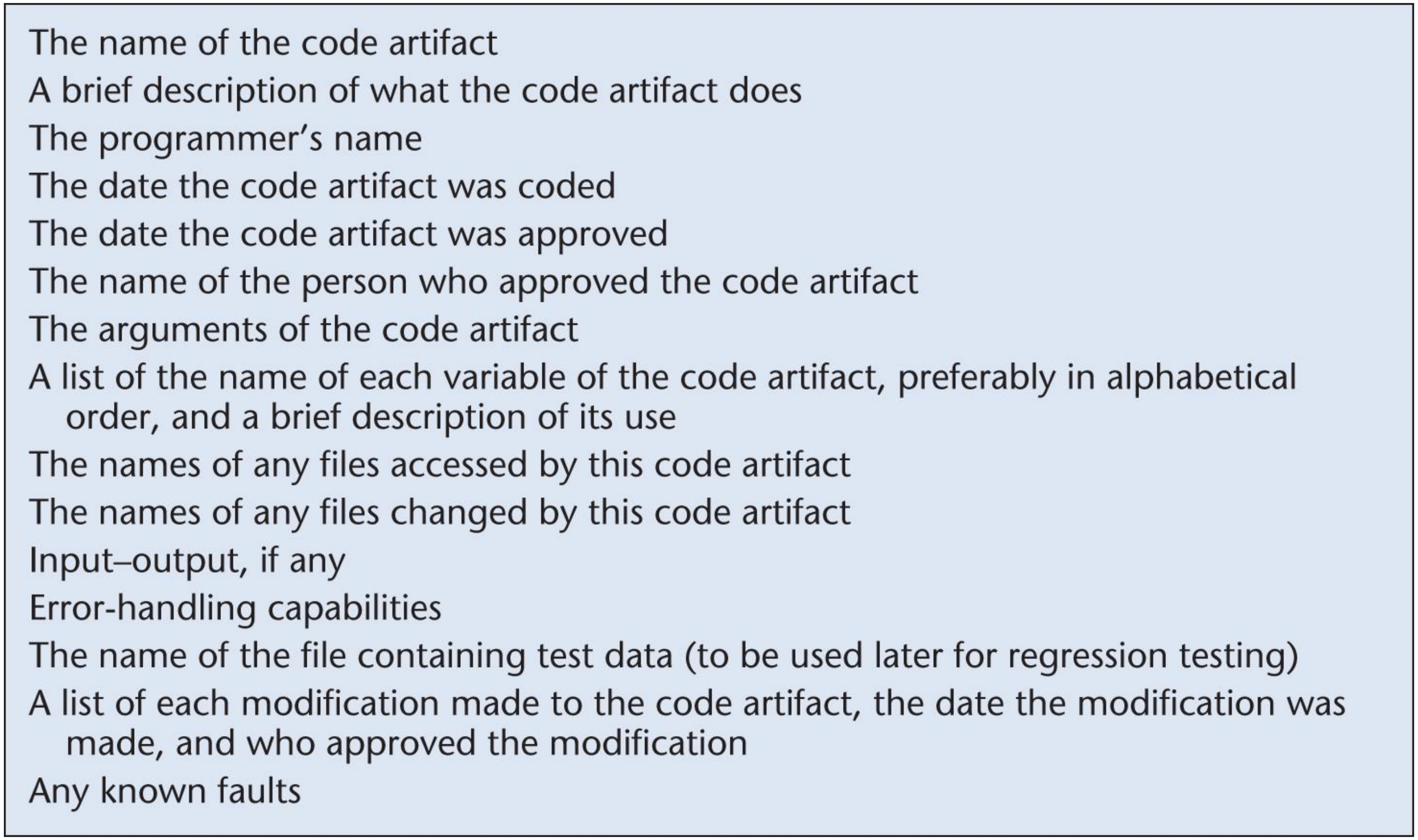
- Self-documenting code is very rare
- The main issue is that the code artifact should be understood easily by
  - Software Quality Assurance team
  - Maintenance team
  - Others who read the code

Self-Documenting Code - Example

- Example:
  - Code artifact contains the variable `xCoordinateOfPositionOfRobotArm`
  - This variable is abbreviated to `xCoord`
  - This is fine, because the entire module deals with the movement of the robot arm
  - But does the maintenance team know this?

Comments for a Code Artifact

- Minimal comments for a code artifact



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

Programming Comments

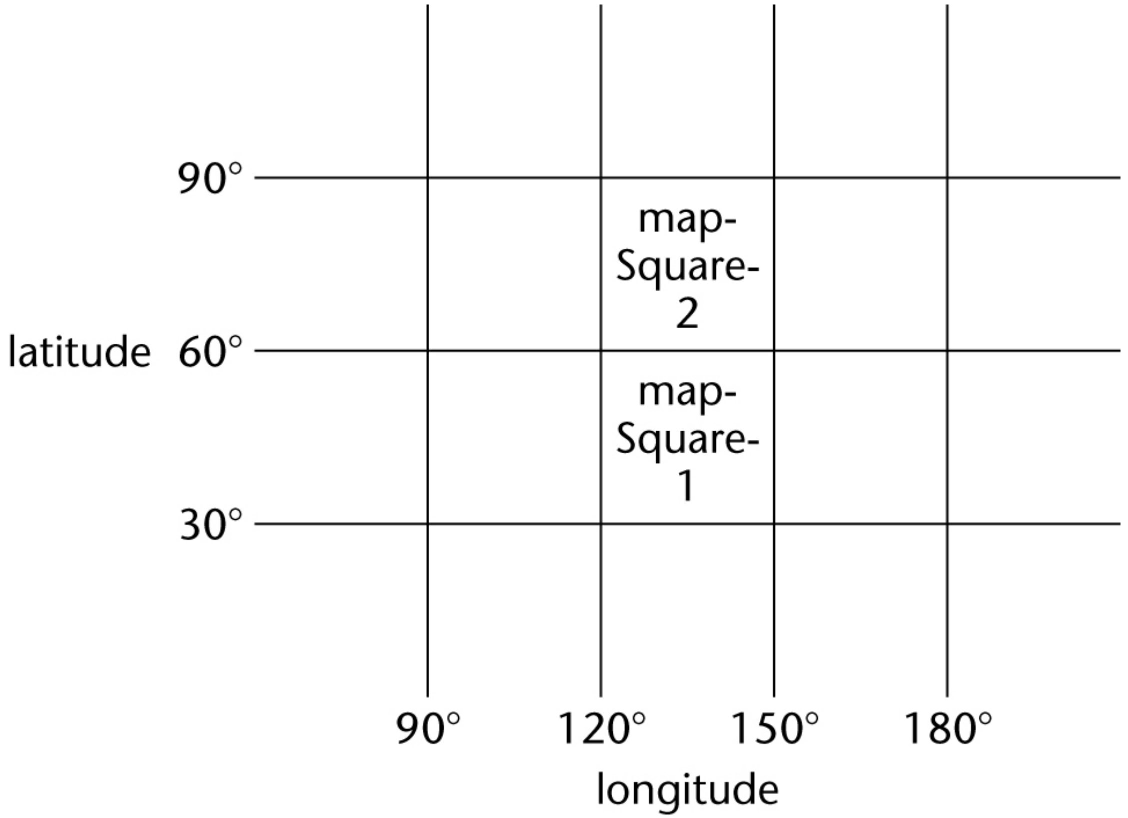
- Comments are essential whenever the code is written in a non-obvious way, or makes use of some indirect aspect of the language
- Comments can help future maintenance team.
- If the comments fail to describe the code, re-code in a clearer way.
- We must never promote/excuse poor programming

Increasing Code Readability

- Use indentation
- Use blank lines to break up big blocks of code

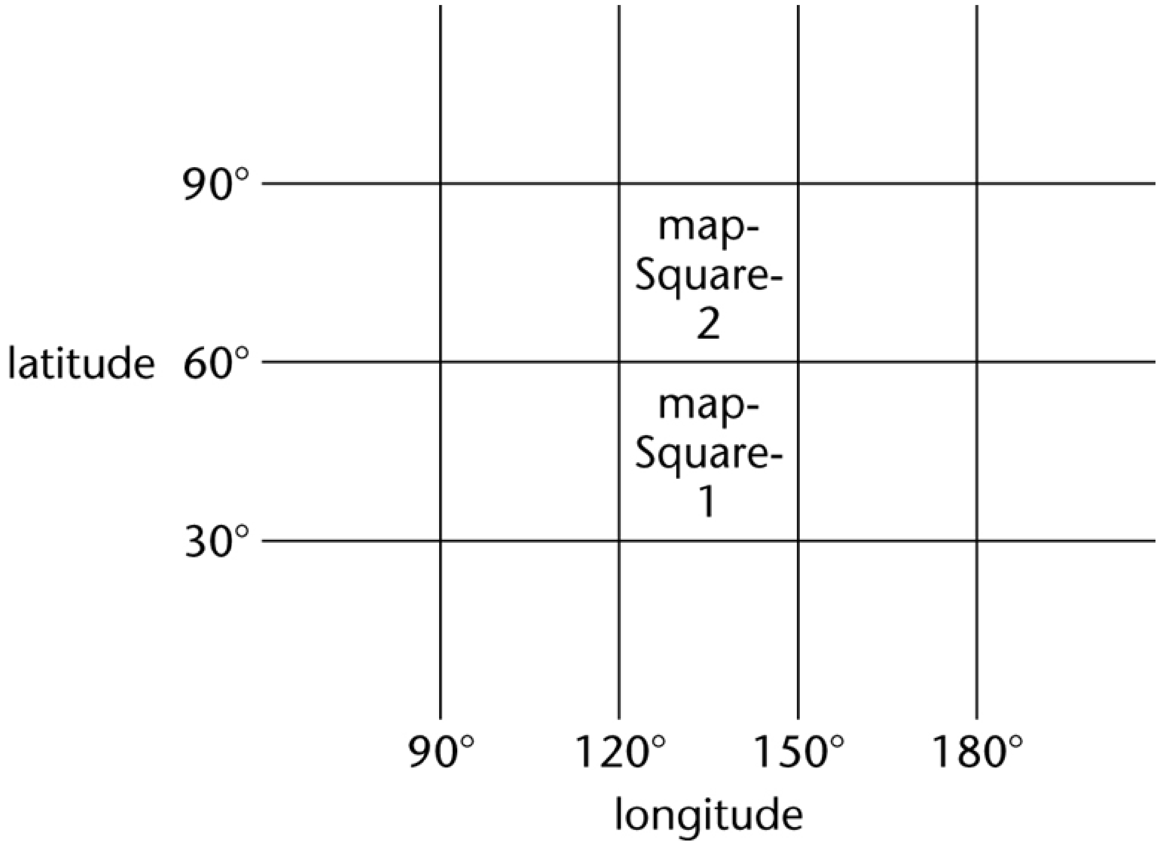
Code Readability - Nested `if` Example

A map consists of two squares. Write code to *determine* whether a point on the Earth's surface lies in `mapSquare1` or `mapSquare2`, or is not on the map

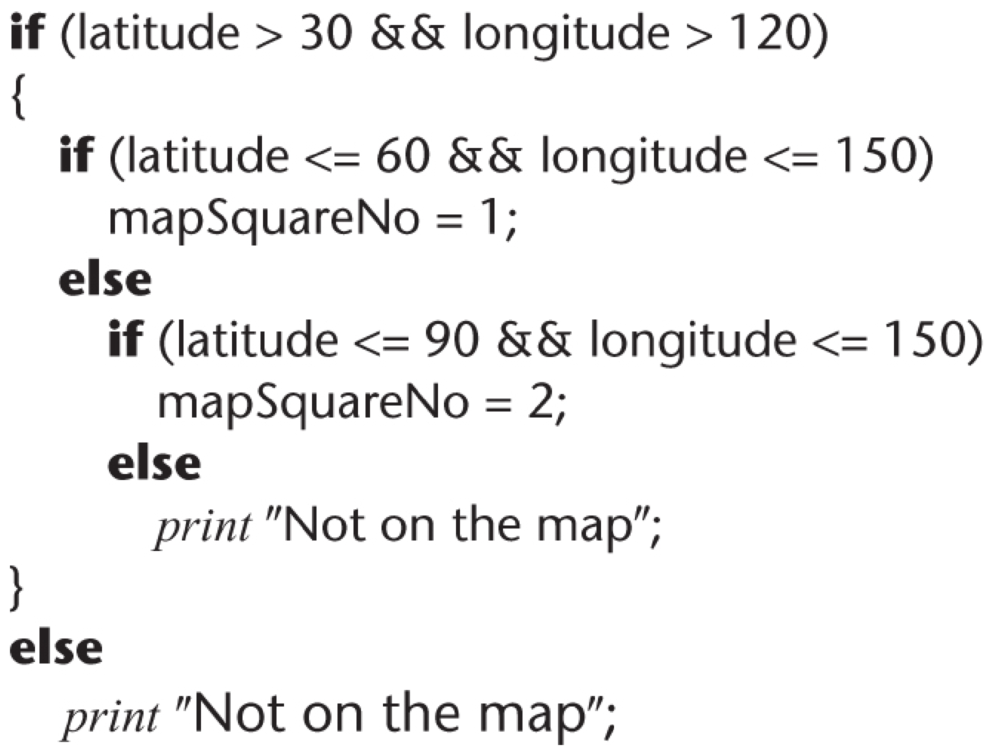


Solution 1 - Badly formatted

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”;



Solution 2: Well-formatted but badly constructed Form



```
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";
```

Solution 3: Accepted Structure

```
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";
```

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>
  ```

  to an equivalent form

  ```
      if <condition1> && <condition2>
  ```
- **Rule of thumb**
  - `if` statements nested to a depth of greater than three should be avoided as poor programming practice

Programming Standards

- Recommendation examples:
  - Nesting of `if` statements should not exceed a depth of level 3
  - Modules should consist of between 35 and 50 statements
  - Use of `goto`s should be avoided. However, a forward `goto` may be used for error handling”

Programming Fault Types

- Faults in the Interface specification
- Difference between the client needs and the product results
- Divergence between requirements and implementation
- Algorithmic Faults
- Missing of variables initialization
- Branching errors (too early or, too late)
- Missing test for null values

Classical Programming Errors

- Use of uninitialized variables
- Jumps into loops
- Non-terminating loops
- Incompatible assignment
- Array indices out-of-bound
- Incorrect logical or procedure
- Improper modification of loop variable

Software Errors Detection

- **Error detection** (while system is running):
  - Testing: Create failures in a planned way
  - Debugging: Start with an unplanned failures
  - Monitoring: Deliver information about product state(s).
  - Find performance bugs.

Software Errors Prevention

- **Error prevention** (before the system is released):
  - Use good programming methodology to reduce complexity
  - Use version control to prevent inconsistent system
  - Apply verification to prevent algorithmic bugs

Software Errors Recovery

- **Error recovery** (recover from failure once the system is released):
  - Database systems (atomic transactions)
  - Modular redundancy
  - Recovery blocks