SOFTWARE ENGINEERING

CO3001

CHAPTER 7.3 – MORE ON IMPLEMENTATION

Adapted from E.J. Braude (2001), Software Engineering: An Object-Oriented Perspective, ISBN 978-0-471-32208-5, John Wiley.

WEEK 10

TOPICS COVERED

- ✓ Implementation meaning
- ✓ Coding style & standards
- ✓ Code with correctness justification
- ✓ Integration meaning
- ✓ Integration process

IMPLEMENTATION

smallest part that will be separately maintained

- Implementation = Unit Implementation + Integration

put them all together

GOLDEN RULE (!?)

- ✓ Requirements to satisfy Customers
- ✓ Design again requirements only
- ✓ Implement again design only
- ✓ Test again design and requirements

IMPLEMENT CODE

- ✓ 1. Plan the structure and residual design for your code
- ✓ 2. Self-inspect your design and/or structure
- ✓ 3. Type your code
- ✓ 4. Self-inspect your code
- ✓ 5. Compile your code
- ✓ 6. Test your code

GENERAL PRINCIPLES IN PROGRAMMING PRACTICE

- ✓ 1. Try to re-use first
- ✓ 2. Enforce intentions
  - If your code is intended to be used in particular ways only, write it so that the code cannot be used in any other way.
  - If a member is not intended to be used by other functions, enforce this by making it private or protected etc.
  - Use qualifiers such as final and abstract etc. to enforce intentions

"THINK GLOBALLY, PROGRAM LOCALLY"

- ✓ Make all members
  - as local as possible
  - as invisible as possible
    - attributes private:
      - access them through more public accessor functions if required.
      - (Making attributes protected gives objects of subclasses access to members of their base classes -- not usually what you want)

EXCEPTIONS HANDLING

"If you must choice between throwing an exception and continuing the computation, continue if you can" (Cay Horstmann)

- ✓ Catch only those exceptions that you know how to handle
- ✓ Be reasonable about exceptions callers must handle
- ✓ Don't substitute the use of exceptions for issue that should be the subject of testing

NAMING CONVENTIONS

- ✓ Use concatenated words
  - e.g., cylinderLength
- ✓ Begin class names with capitals
- ✓ Variable names begin lower case
- ✓ Constants with capitals
  - as in MAX_N or use static final
- ✓ Data members of classes with an underscore
  - as in _timeOfDay
- ✓ Use get…, set…., and is… for accessor methods
- ✓ Additional getters and setters of collections
- ✓ And/or distinguish between instance variables, local variables and parameters

DOCUMENTING METHODS

- ✓ what the method does
- ✓ why it does so
- ✓ what parameters it must be passed (use @param tag)
- ✓ exceptions it throws (use @exception tag)
- ✓ reason for choice of visibility
- ✓ known bugs
- ✓ test description, describing whether the method has been tested, and the location of its test script
- ✓ history of changes if you are not using a CM system
- ✓ example of how the method works
- ✓ pre- and post-conditions
- ✓ special documentation on threaded and synchronized methods

```
/* Class Name          : EncounterCast
* Version information   : Version 0.1
* Date                  : 6/19/1999
* Copyright Notice      : see below
* Edit history:
*   11 Feb
*    8 Feb
*   08 Jan
*/

/*
    Copyri

    This pr
    "Softwa
    by Eric
```

```
/** Facade class/object for the EncounterCharacters package. Used to
* reference all characters of the Encounter game.
* <p> Design: SDD 3.1 Module decomposition
* <br> SDD 5.1.2 Interface to the EncounterCharacters package
*
* <p>Design issues:<ul>
* <li> SDD 5.1.2.4 method engagePlayerWithForeignCharacter was
*   not implemented, since engagements are handled more directly
*   from the Engaging state object.
* </ul
*
* @a
* @v
*/
publi
{
    /** Name for human player */
    private static final String MAIN_PLAYER_NAME = "Elena";
```

```
/** Gets encounterCast, the one and only instance of EncounterCast.
     * <p> Requirement: SDD 5.1.2
     *
     * @return      The EncounterCast singleton.
     */
     public static EncounterCast getEncounterCast()
        { return encounterCastS; }
```

DOCUMENTING ATTRIBUTES

- ✓ Description -- what it's used for
- ✓ All applicable invariants
  - quantitative facts about the attribute,
    - such as "1 < _age < 130"
    - or " 36 < _length * _width < 193".

CONSTANTS

- Before designating a final variable, be sure that it is, indeed, final. You're going to want to change "final" quantities in most cases. Consider using method instead.
- Ex:
  - instead of ...
  - protected static final MAX_CHARS_IN_NAME;
  - consider using ...
  - protected final static int getMaxCharsInName()
  - { return 20;
  - }

INITIALIZING ATTRIBUTES

- Attributes should be always be initialized, think of
  - private float _balance = 0;
- Attribute may be an object of another class, as in
  - private Customer _customer;
- Traditionally done using the constructor, as in
  - private Customer _customer = new Customer("Edward", "Jones");
- Problem is maintainability. When new attributes added to Customer, all have to be updated. Also accessing persistent storage unnecessarily.

INSPECT CODE 1 OF 5: CLASSES OVERALL

- ✓ C1. Is its (the class') name appropriate?
- ✓ C2. Could it be abstract (to be used only as a base)?
- ✓ C3. Does its header describe its purpose?
- ✓ C4. Does its header reference the requirements and/or design element to which it corresponds?
- ✓ C5. Does it state the package to which it belongs?
- ✓ C6. Is it as private as it can be?
- ✓ C7. Should it be final (Java)
- ✓ C8. Have the documentation standards been applied?

INSPECT CODE 2 OF 5 : ATTRIBUTES

- ✓ A1. Is it (the attribute) necessary?
- ✓ A2. Could it be static?
- ✓ A3. Should it be final?
- ✓ A4. Are the naming conventions properly applied?
- ✓ A5. Is it as private as possible?
- ✓ A6. Are the attributes as independent as possible?
- ✓ A7. Is there a comprehensive initialization strategy?

INSPECT CODE 3 OF 5 : CONSTRUCTORS

- ✓ CO1. Is it (the constructor) necessary?
- ✓ CO2. Does it leverage existing constructors?
- ✓ CO3. Does it initialize of all the attributes?
- ✓ CO4. Is it as private as possible?
- ✓ CO5. Does it execute the inherited constructor(s) where necessary?

INSPECT CODE 4 OF 5: METHOD HEADERS

- ✓ MH1. Is the method appropriately named?
- ✓ MH2. Is it as private as possible?
- ✓ MH3. Could it be static?
- ✓ MH4. Should it be be final?
- ✓ MH5. Does the header describe method's purpose?
- ✓ MH6. Does the method header reference the requirements and/or design section that it satisfies?
- ✓ MH7. Does it state all necessary invariants? (section 4)
- ✓ MH8. Does it state all pre-conditions?
- ✓ MH9. Does it state all post-conditions?
- ✓ MH10.Does it apply documentation standards?
- ✓ MH11.Are the parameter types restricted? (see section 2.5)

INSPECT CODE 5 OF 5: METHOD BODIES

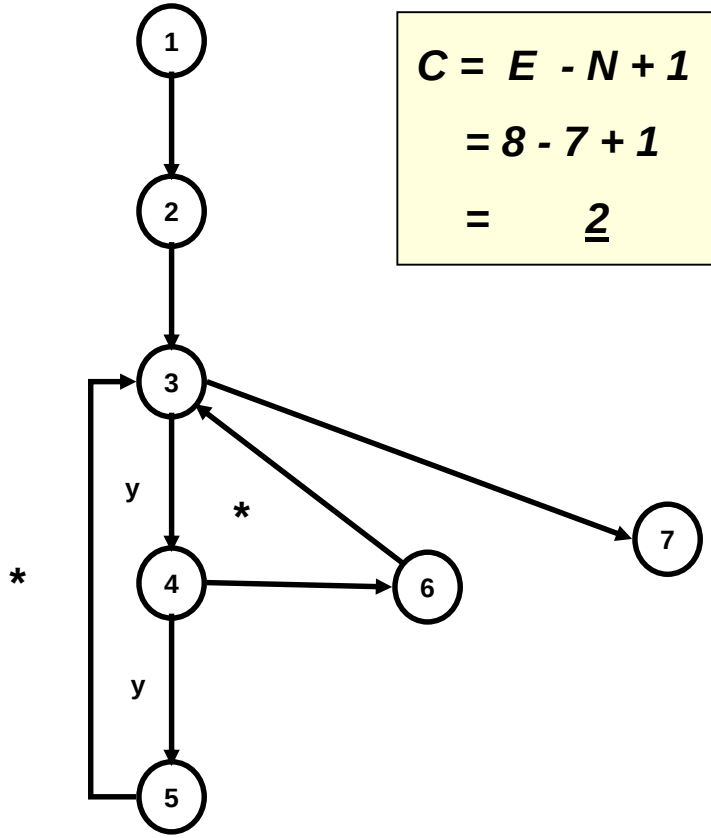
- ✓ MB1. Is the algorithm consistent with the detailed design pseudocode and/or flowchart?
- ✓ MB2. Does the code assume no more than the stated preconditions?
- ✓ MB3. Does the code produce every one of the postconditions?
- ✓ MB4. Does the code respect the required invariant?
- ✓ MB5. Does every loop terminate?
- ✓ MB6. Are required notational standards observed?
- ✓ MB7. Has every line been thoroughly checked?
- ✓ MB8. Are all braces balanced?
- ✓ MB9. Are illegal parameters considered? (see section 2.5)
- ✓ MB10. Does the code return the correct type?
- ✓ MB11. Is the code thoroughly commented?

STANDARD METRICS FOR SOURCE CODE

- ✓ Counting lines
  - Lines of code (LoC)
    - How to count statements that occupy several lines (1 or n?)
    - How to count comments (0?)
    - How to count lines consisting of while, for, do, etc. (1?)
- ✓ IEEE metrics
  - 14. Software Science Measures
    - n1, n2 = num. of distinct operators (+,* etc.), operands
    - N1, N2 = total num. of occurrences of the operators, the operands
    - Estimated program length = n1(log n1) + n2(log n2)
    - Program difficulty = (n1N1)/(2n2)
  - 16. Cyclomatic Complexity
    - …
- ✓ Custom metrics?

CYCLOTOMIC COMPLEXITY

```
1  int x = anX;
2  int y = aY;
3  while( !( x == y) ) {
4          if( x > y )
5                  x = x - y;
           else
6                  y = y - x;
7  }
8  ...println( x );
```



$C = E - N + 1$

$= 8 - 7 + 1$

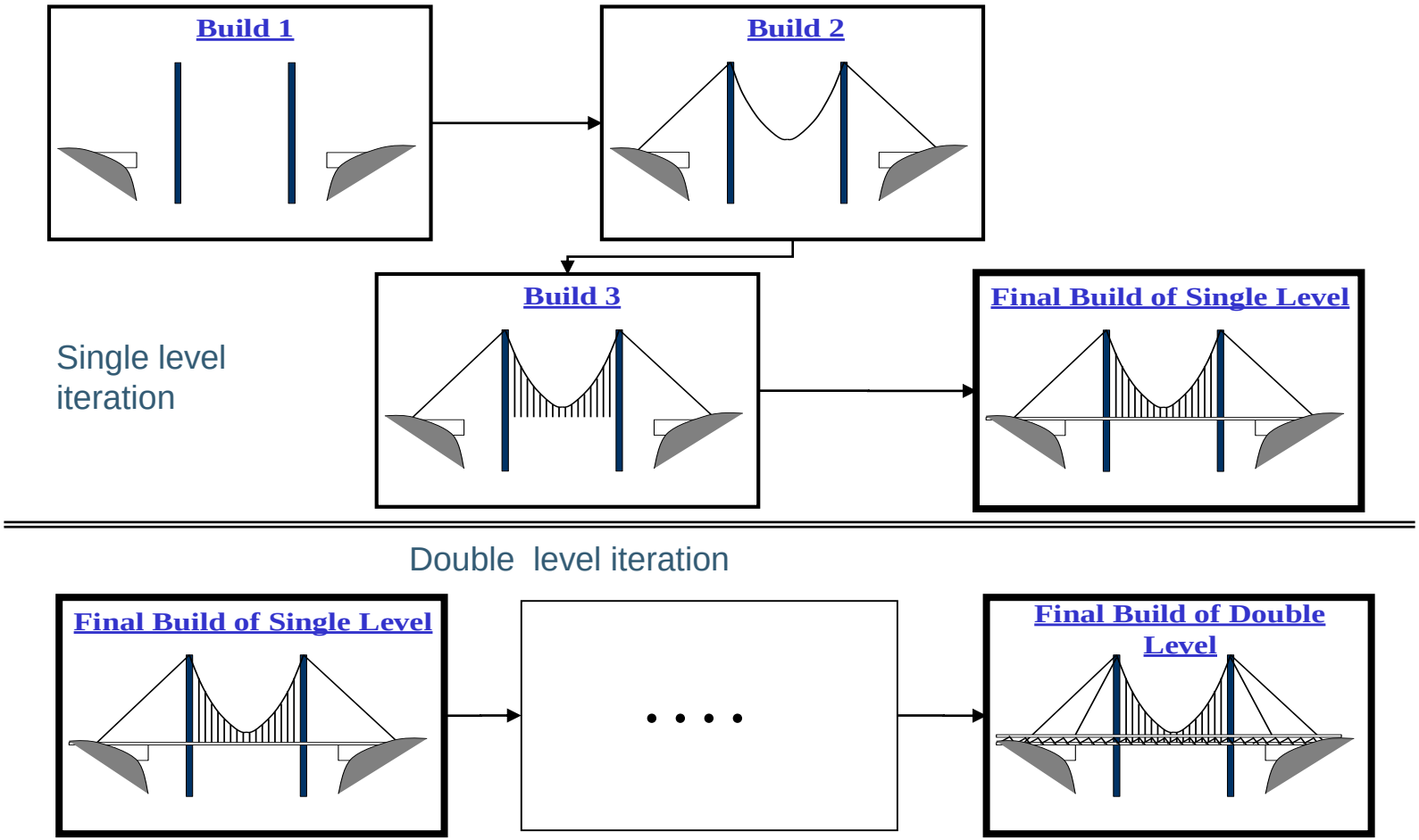
$= 2$

* : independent loop

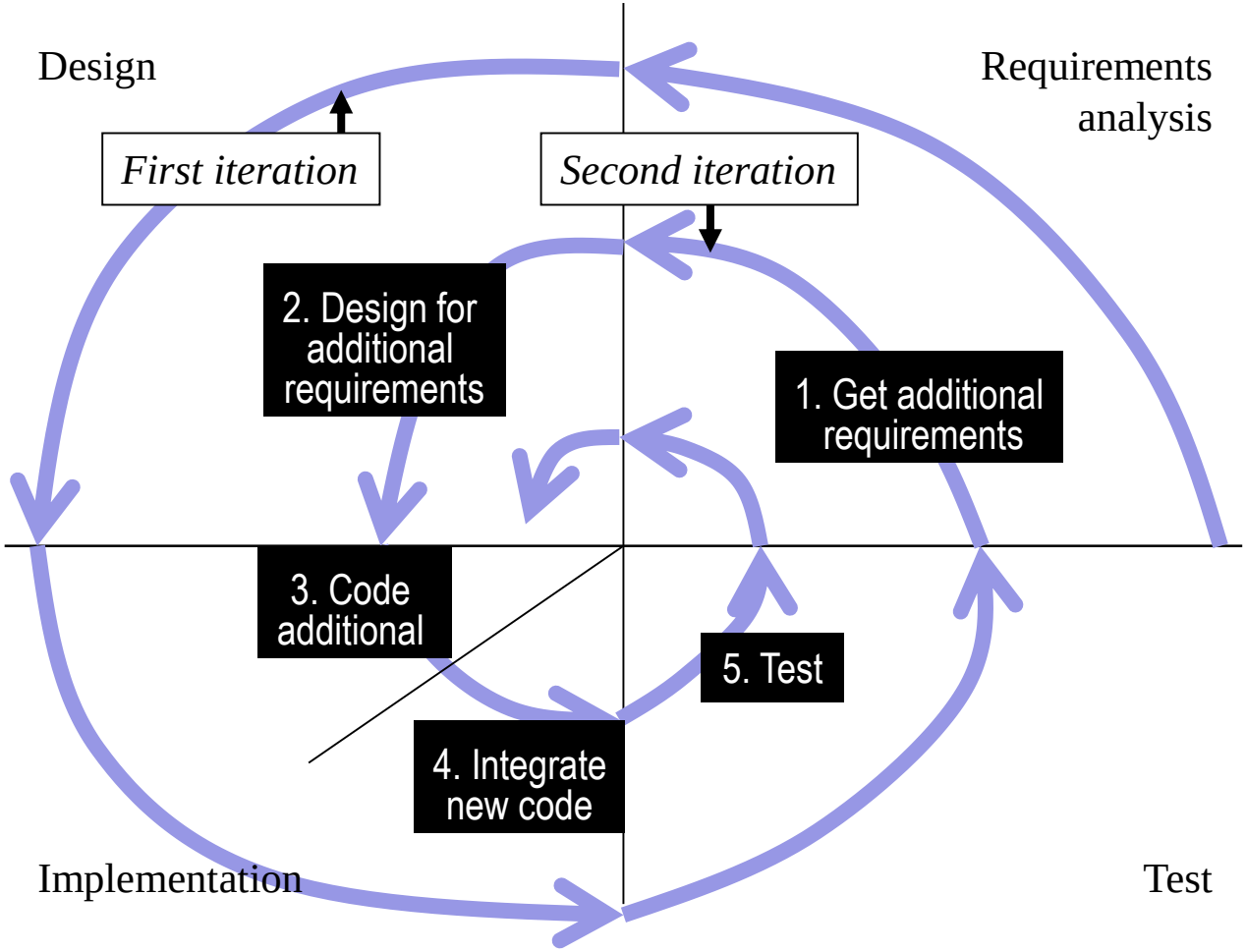
INTEGRATION

- ✓ Applications are complex => be built of parts => assembled: integration
- ✓ Waterfall process
  - Integration phase is (nearly) the last
  - Incompatibility ?

THE BUILD PROCESS



INTEGRATION IN SPIRAL DEVELOPMENT



ROADMAP FOR INTEGRATION AND SYSTEM TEST

1. Decide extent of all tests.

2. For each iteration …

2.1 For each build …

- **2.1.1 Perform regression testing from prior build**
- **2.1.2 Retest functions if required**
- **2. 1.3 Retest modules if required**
- **2. 1.4 Test interfaces if required**
- **2. 1.5 Perform build integration tests**

Development of iteration complete

2.2 Perform iteration system and usability tests

System implemented

3. Perform installation tests

System installed

4. Perform acceptance tests

Job completed

FACTORS DETERMINING THE SEQUENCE OF INTEGRATION

- ✓ Technical:
  - Usage of modules by other modules
    - build and integrate modules used before modules that use them
  - Defining and using framework classes
- ✓ Risk reduction:
  - Exercising integration early
  - Exercising key risky parts of the application as early as possible
- ✓ Requirements:
  - Showing parts or prototypes to customers

SUMMARY

- ✓ Keep coding goals in mind:
  - 1. correctness
  - 2. clarity
- ✓ Apply programming standards
- ✓ Specify pre- and post-condition
- ✓ Prove programs correct before compiling
- ✓ Track time spent
- ✓ Maintain quality and professionalism
- ✓ Integration process executed in carefully planned builds