SOFTWARE ENGINEERING

Chapter 8 - Implementation

TUẦN 13

Topics covered

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

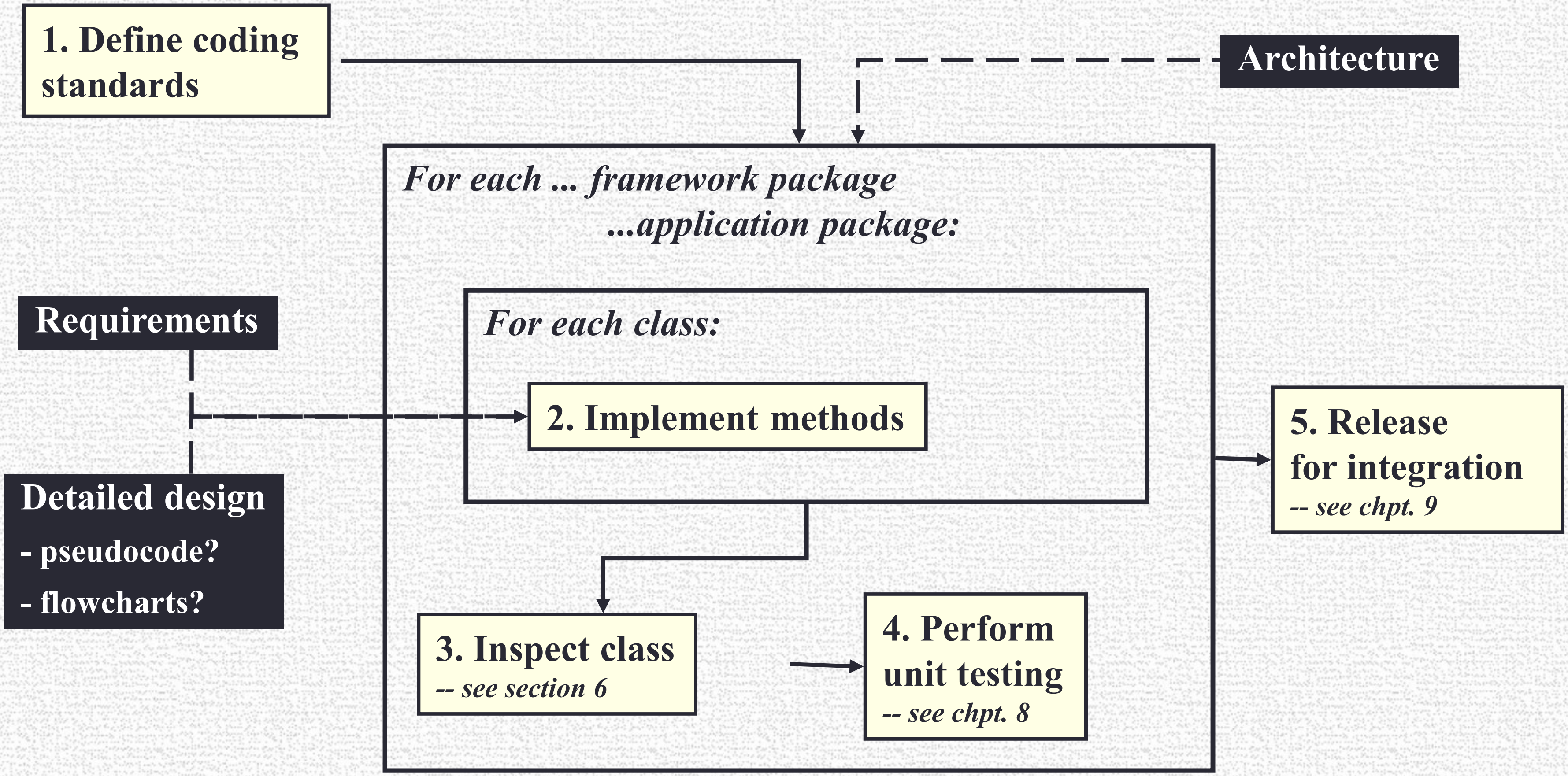
Implementation

- Implementation = Unit Implementation + Integration
  - “Unit Implementation”:
    - “Implementation” = programming
    - “Unit” = smallest part that will be separately maintained.
- Goals:
  - Satisfy the requirements (specified by the detail design)
- Coding goals:
  - Correctness
  - Clarity
  - …?

Golden rule (!?)

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

Roadmap for Unit Implementation

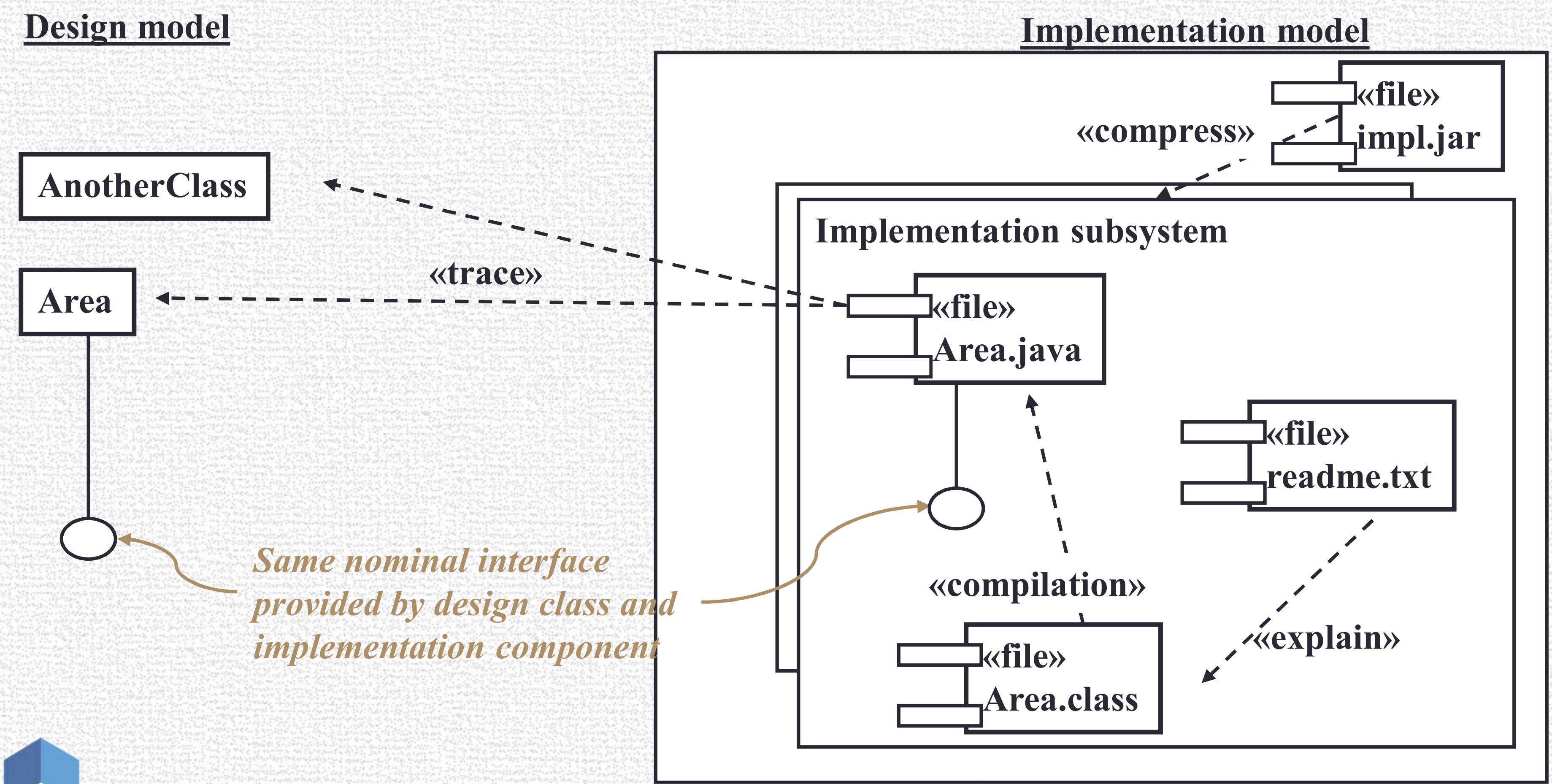


One way to ...

Prepare for Implementation

- 1. Confirm the detailed designs you must implement
  - code only from a written design (part of the SDD)
- 2. Prepare to measure time spent, classified by:
  - residual detailed design; detailed design review; coding; coding review; compiling & repairing syntax defects; unit testing (see chapter 7) & repairing defects found in testing
- 3. Prepare to record defects using a form
  - default: major (requirement unsatisfied), trivial, or neither
  - default: error, naming, environment, system, data, other
- 4. Understand required standards
  - for coding
  - for the personal documentation you must keep
    - see the case study for an example
- 5. Estimate size and time based on your past data
- 6. Plan the work in segments of ± 100 LOC

RUP Implementation Model Constituents



Implement Code 1/2

- 1. Plan the structure and residual design for your code
  - (complete missing detailed design, if any)
  - note pre- and post-conditions
  - note the time spent
- 2. Self-inspect your design and/or structure
  - note time spent, defect type, source (phase), severity
- 3. Type your code
  - do not compile yet
  - try methods listed below
  - apply required standards
  - code in a manner that is easiest to verify
    - use formal methods if appropriate

Implement Code 2/2

- 4. Self-inspect your code -- do not compile yet
  - convince yourself that your code does the required job
    - the compiler will never do this for you: it merely checks syntax!
  - note time spent, defects found, type, source, severity
  - see the code inspection checklist for details commonly required for method & class construction.
- 5. Compile your code
  - repair syntax defects
  - note time spent, defect type, source, severity , and LOC.
- 6. Test your code
  - apply unit test methods in chapter 9
  - note time spent, defects found, type, source, severity

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.

Applications of “Enforce Intentions”

- 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)

Miscellaneous

- Avoid type inquiry
  - e.g. if(x instanceof MyClass)
  - virtual function feature instead
- Use Singleton design pattern if there is to be only one instance of a class
  - e.g. theEncounter

Exceptions Handling

- Catch only those exceptions that you know how to handle
  - or handle part & throw
  - outer scope can do so, e.g.,
    - … myMethod(…) throws XYZException
    - { ...
    - calledFunction(); // throws XYZException
    - …
    - }
- Be reasonable about exceptions callers must handle
- Don’t substitute the use of exceptions for issue that should be the subject of testing
  - e.g. null parameters (most of the time)
- Consider providing
  - a version throwing exceptions, and
  - a version which does not (different name)
    - accompanied by corresponding test functions.
      - e.g., pop empty stack

Exceptions Handling

- "If you must choice between throwing an exception and continuing the computation, continue if you can" (Cay Horstmann)
  - The game should continue with a default name when given a faulty name
  - A banking transaction with an illegal amount would not be allowed to continue

One way to ...

Implement Error Handling

- 1. Follow agreed-upon development process; inspect
- 2. Consider introducing classes to encapsulate legal parameter values
  - private constructor; factory functions to create instances
  - catches many errors at compile-time
- 3. Where error handling is specified by requirements, implement as required
  - use exceptions if passing on error handling responsibility
- 4. For applications that must never crash, anticipate all possible implementation defects (e.g., use defaults)
  - only if unknown performance better than none (unusual!)
- 5. Otherwise, follow a consistent policy for checking parameters
  - rely mostly on good design and development process

Naming Conventions

- Use concatenated words
  - e.g., cylinderLength
- Begin class names with capitals
- Variable names begin lower case
- Constants with capitals
  - as in MAX_NAME_LENGTH
  - use static final
    - but consider method instead
- Data members of classes with an underscore
  - as in _timeOfDay
  - or equivalent
  - to distinguish them from other variables
    - since they are global to their object
- Use get…, set…., and is… for accessor methods
  - as in getName(), setName(), isBox()
    - latter returns boolean

Naming Conventions (cont.)

- Additional getters and setters of collections
  - e.g., insertIntoName(), removeFromName().
- Consider preceding with standard letters or combinations of letters
  - e.g., C….. for classes
    - as in CCustomer etc.
  - useful when the importance of knowing the types of names exceeds the awkwardness of strange-looking names.
  - or place these type descriptors at the end
- And/or distinguish between instance variables, local variables and parameters
  - _length, length and aLength

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 2000     Tom VanCourt    Used simplified test interface.
*   8 Feb
*   08 Jar
*/

/*
   Copyri

   This pr
   "Softw
   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 implemen
*   from the Enga
* </ul>
*
* @author   Dr. E
* @version  0.1
*/
public class Enc
{
   /** 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;

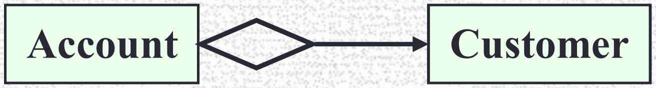
  - consider using ...
  - protected final static int getMaxCharsInName()
  - { return 20;
  - }

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.

One Solution to Object Initialization

Use initialization when the value is first accessed. Supply **MyClass** with static **getDefaultMyClass()**. Attributes are declared without initialization, then assigned values the first time they are accessed.



In class Customer:

```
….
public static Customer getDefaultCustomer()
// … reasons these values are chosen for the default
{  return new Customer ( "John", "Doe", 0, 1000, -2000 );
}
```

```
public Customer getCustomer()   { // access customerI
  if( customerI == null )       // never accessed
    customerI = Customer.getDefaultCustomer(); // initial value
  return customerI;             // current value
}

public getDefaultAccount()      // for users of Account
{  return new Account( -10, 3, "regular" );
}
```

In class Account:

```
….
private float balanceI = -10;
private Customer customerI;

public Account( ….. )….

public float getBalance()
{              return  balanceI;
}
```

Inspect Code 1 of 5: Classes Overall

One way to ...

- C1. Is its (the class') name appropriate?
  - consistent with the requirements and/or the design?
  - sufficiently specialized / general?
- 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?
  - e.g., Javadoc

Inspect Code 2 of 5 : Attributes

One way to ...

- A1. Is it (the attribute) necessary?
- A2. Could it be static?
  - Does every instance really need its own variable?
- A3. Should it be final?
  - Does its value really change?
    - Would a "getter" method alone be preferable (see section tbd)
- 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?
  - at declaration-time?
  - with constructor(s)?
  - using static{}?
  - Mix the above? How?

Inspect Code 3 of 5 : Constructors

One way to ...

- CO1. Is it (the constructor) necessary?
  - Would a factory method be preferable?
    - More flexible
    - Extra function call per construction
- CO2. Does it leverage existing constructors?
  - (a Java-only capability)
- 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

One way to ...

- MH1. Is the method appropriately named?
  - method name consistent with requirements &/or design?
- 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

One way to ...

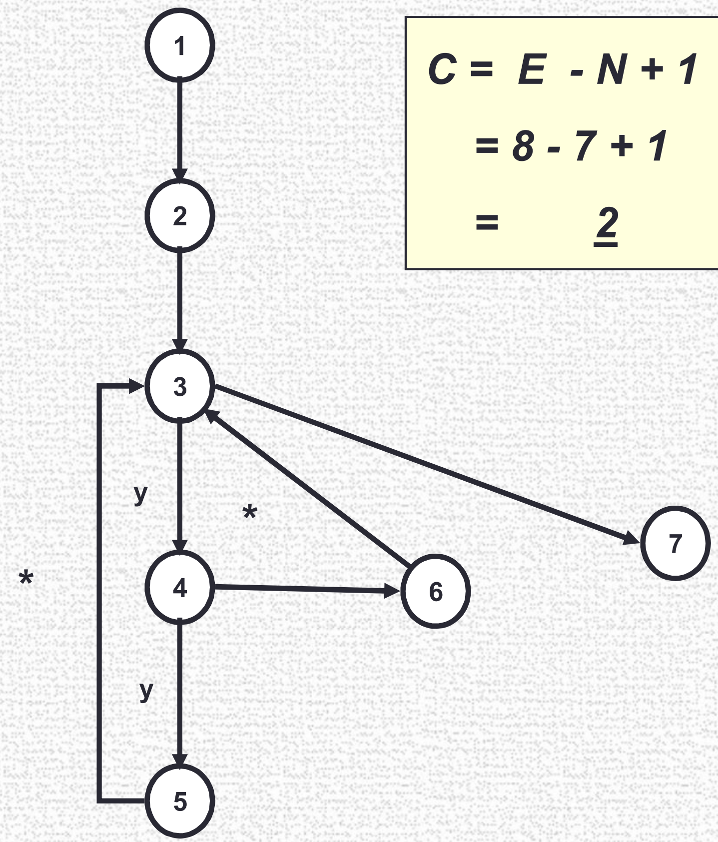
- MB1. Is the algorithm consistent with the detailed design pseudocode and/or flowchart?
- MB2. Does the code assume no more than the stated preconditions?
- MB3. Does the code produce every one of the postconditions?
- MB4. Does the code respect the required invariant?
- MB5. Does every loop terminate?
- 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 );
```



$$C = E - N + 1$$
$$= 8 - 7 + 1$$
$$= \underline{2}$$

* : independent loop

Code Inspection

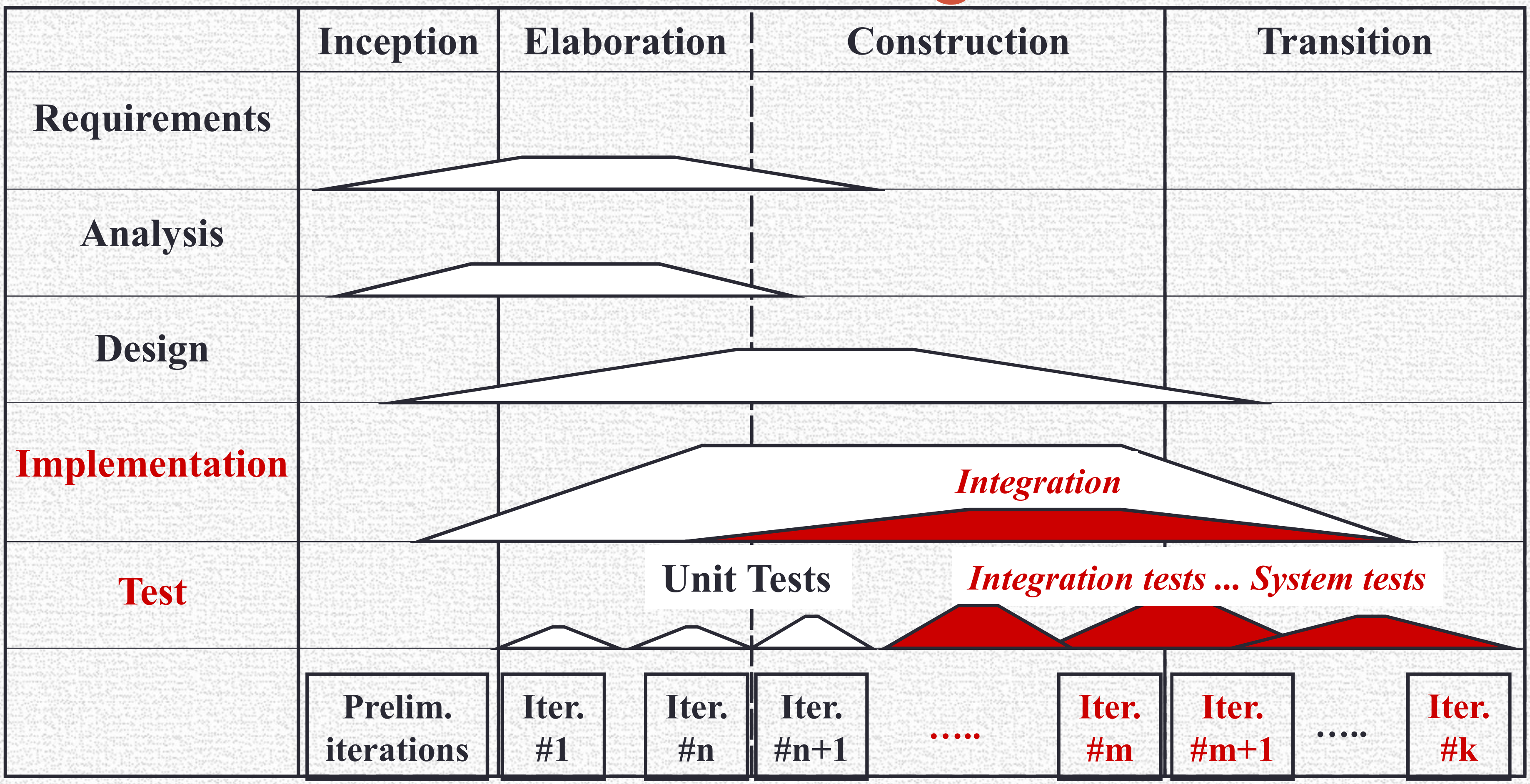
Table 7.1 (6.3) Defect severity classification using triage [3]

Severity	Description
Major	Requirement(s) not satisfied
Medium	Neither major nor trivial
Trivial	A defect which will not affect operation or maintenance

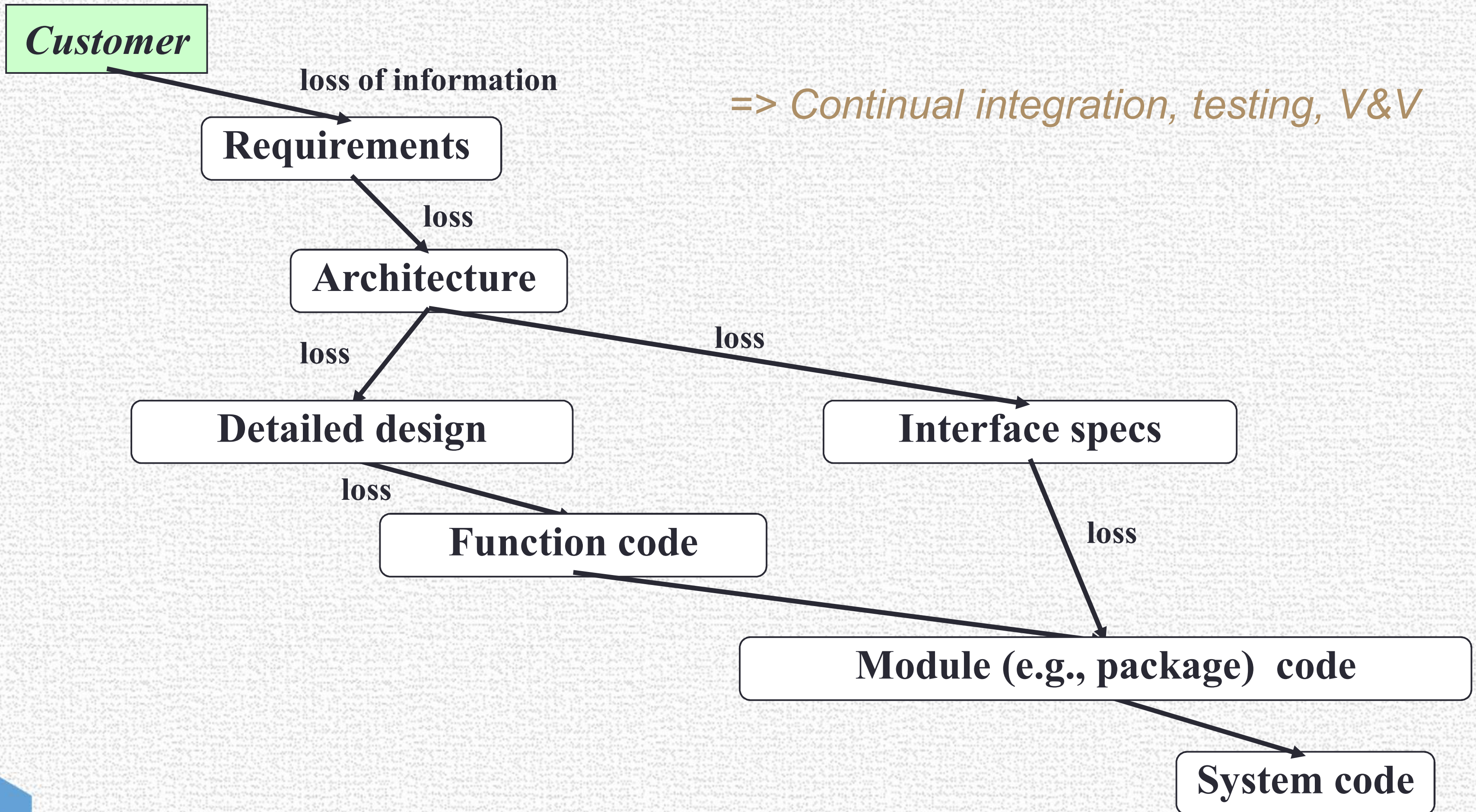
Integration

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

Unified Process for Integration & Test

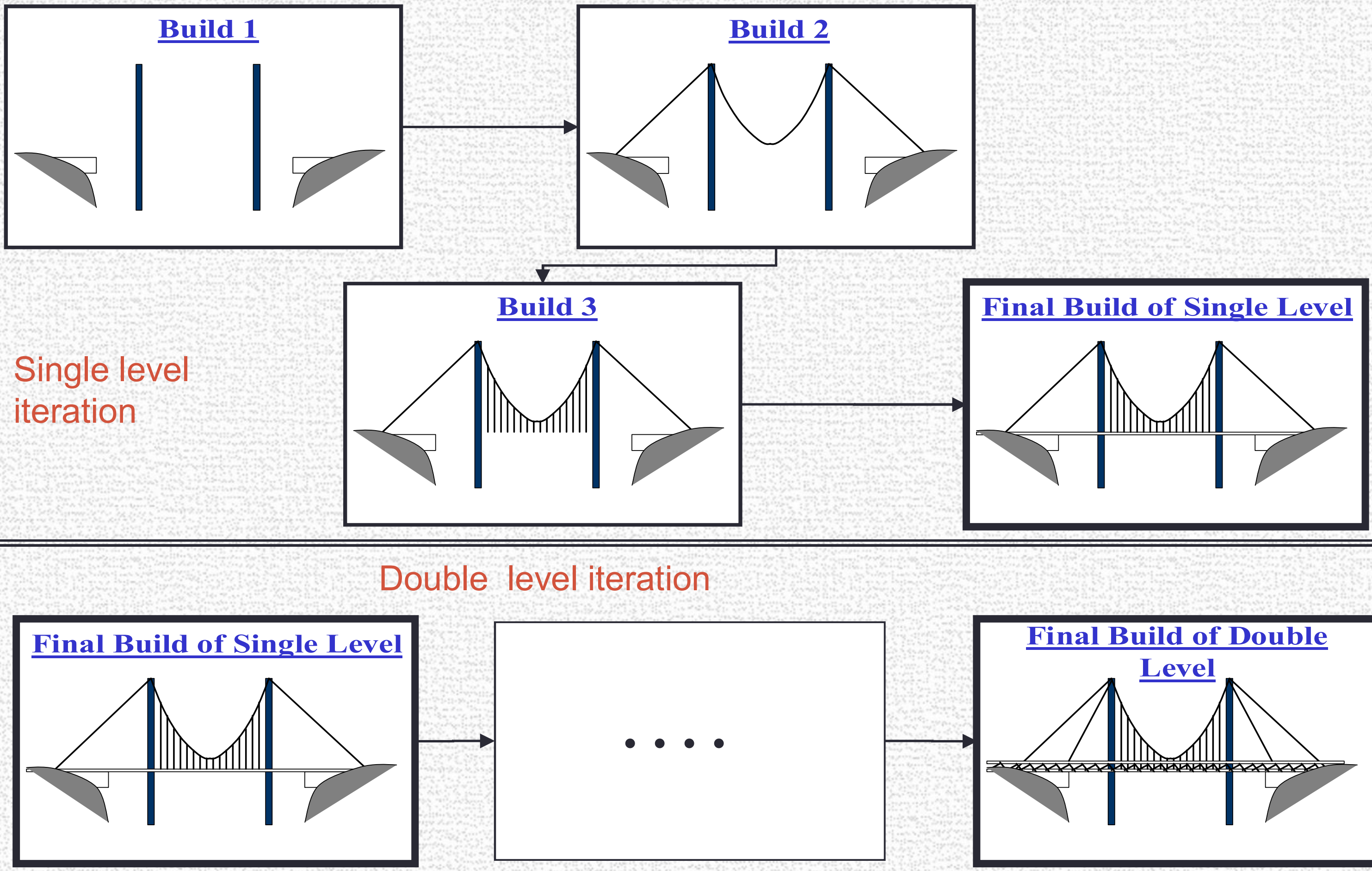


Development Overview

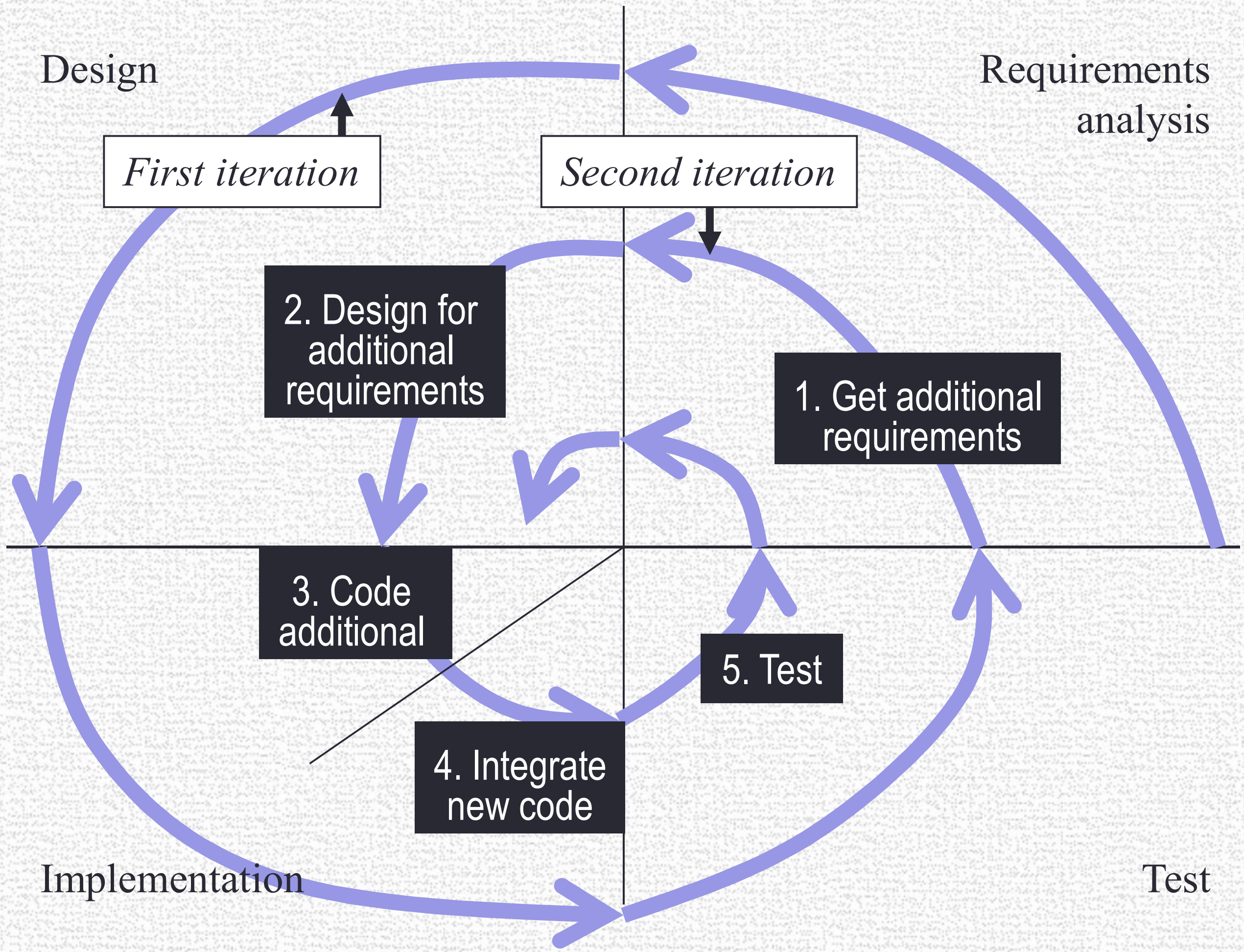


=> Continual integration, testing, V&V

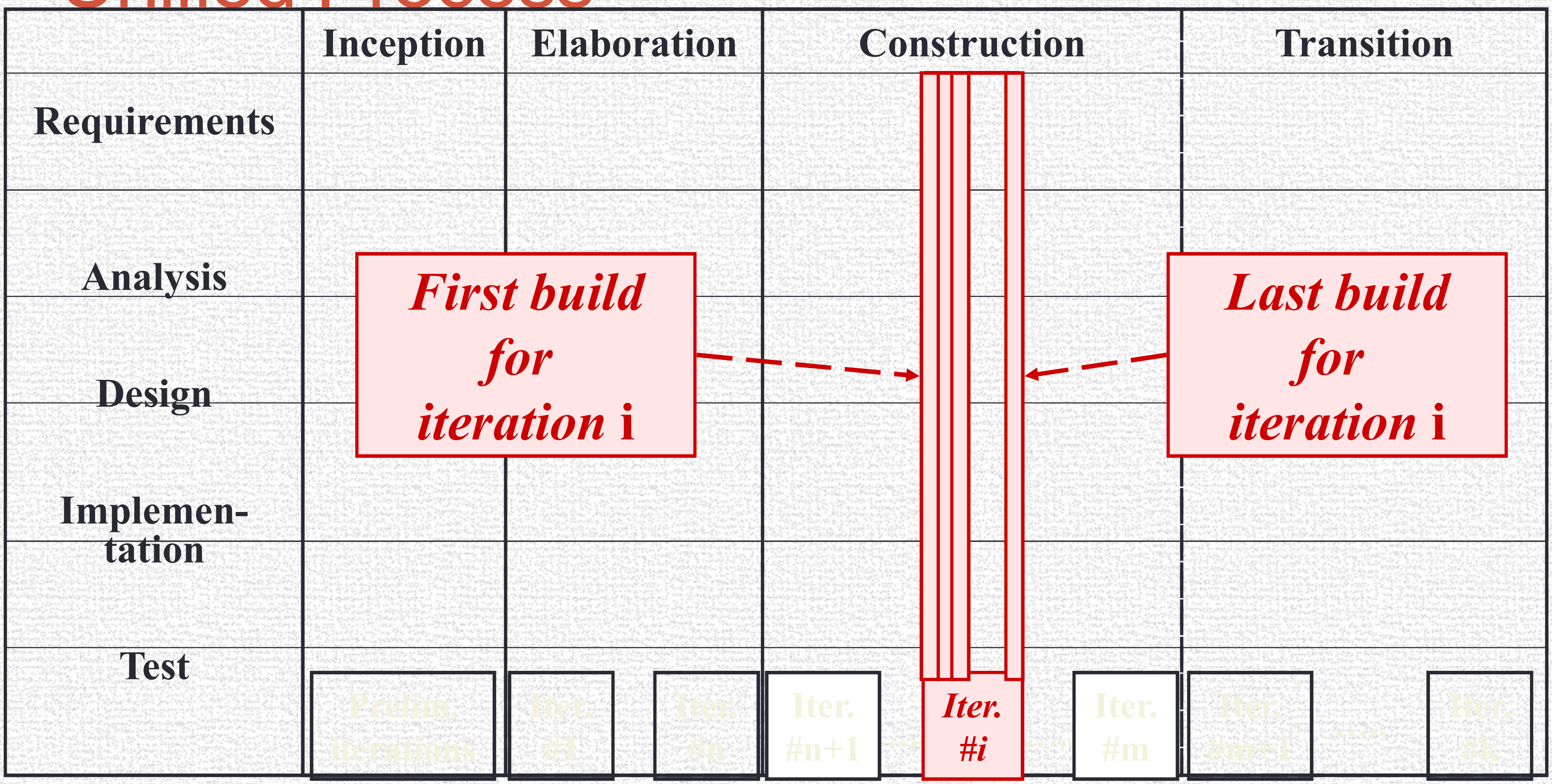
The Build Process



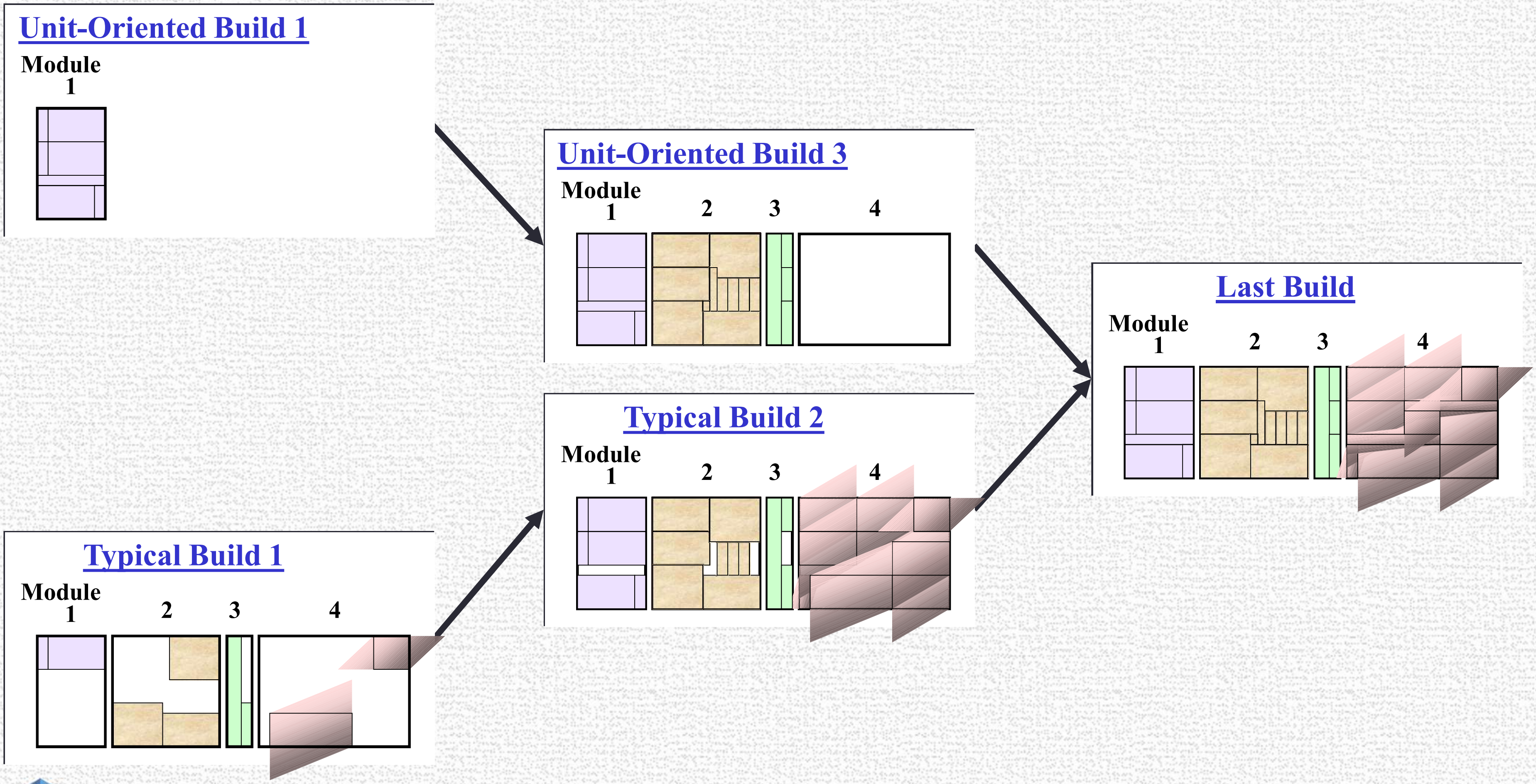
Integration in Spiral Development



Relating Builds and Iterations in the Unified Process



Build Sequences: Ideal vs. Typical



Plan Integration & Builds

One way to …

- 1. Understand the architecture decomposition.
  - try to make architecture simple to integrate
- 2. Identify the parts of the architecture that each iteration will implement.
  - build framework classes first, or in parallel
  - if possible, integrate "continually"
  - build enough UI to anchor testing
  - document requirements for each iteration
  - try to build bottom-up
    - so the parts are available when required
  - try to plan iterations so as to retire risks
    - biggest risks first
  - specify iterations and builds so that each use case is handled completely by one
- 3. Decompose each iteration into builds if necessary.
- 4. Plan the testing, review and inspection process.
  - see section tbd.
- 5. Refine the schedule to reflect the results.

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** -- *section 3.1*

Development of iteration complete

2.2 Perform iteration system and usability tests -- *sections 3.4, 3.5*

System implemented

3. Perform installation tests -- *section 3.8*

System installed

4. Perform acceptance tests -- *section 3.7*

Job complete

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