UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4351 Fall 2023

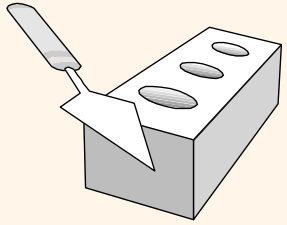
Software Engineering

M & W 4 to 5:30 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



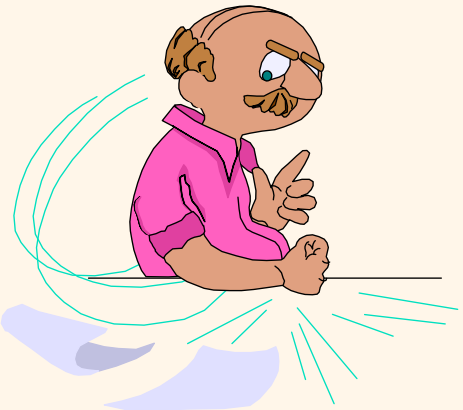
COSC 4351

4 to 5:30

PLEASE

LOG IN

CANVAS



Youyi [A-L]

Kevin [M-Z]

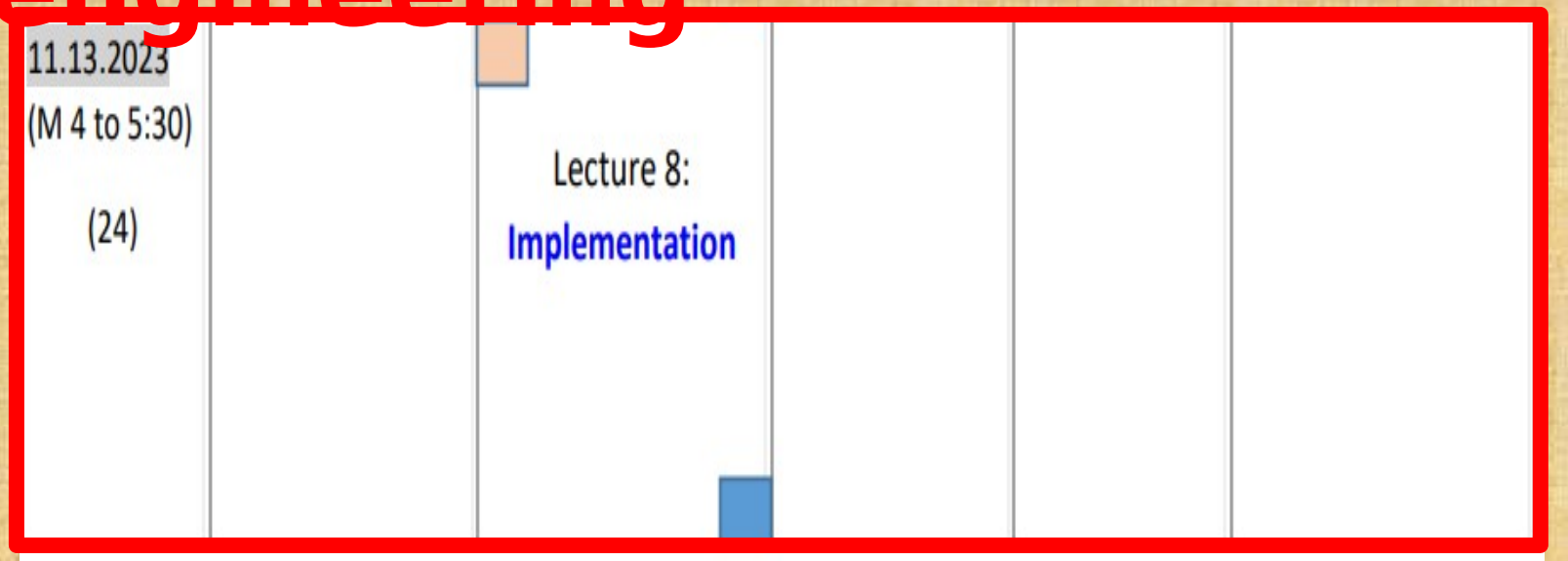
Please close all other windows.

11.13.2023 (M 4 to 5:30) (24)		Lecture 8: Implementation			
11.15.2023 (W 4 to 5:30) (25)		Lecture 9: Testing Tutorial 6 TDD			
11.20.2023 (M 4 to 5:30) (26)		EXAM 4 REVIEW (CANVAS)	Download ZyBook: Sections 12-14		
11.27.2023 (M 4 to 5:30) Optional (27)					Q & A Set 4 topics.
11.29.2023 (W 4 to 5:30) (28) LAST CLASS					EXAM 4 (CANVAS)

Class 24

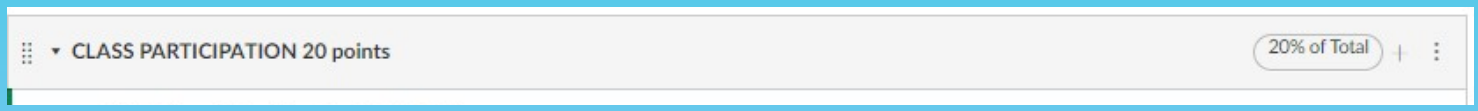
COSC 4351

Software engineering



From 4:00 to 4:10 PM – 10 minutes.

11.13.2023 (M 4 to 5:30) (24)		Lecture 8: **Implementation**			



PASSWORD: IMPLEMENTATION

There is an article on Wall Street Journal today about students cheating while taking final exam at home. I am including two quotes.

The U.S. Military Academy at West Point this month concluded investigations into its largest cheating scandal in at least four decades. It punished dozens of cadets found to be dishonest on an exam while studying remotely.

The U.S. Air Force Academy has said it suspects that 249 cadets cheated during last year's spring semester, with a majority confessing and placed on six-month probation.

NewsBreak
Used by over 45 million people

Open APP

Texas Supreme Court Concludes State Universities May Revoke Degrees By Former Students For Academic Dishonesty

brownwoodnews.com

I found this on NewsBreak: Texas Supreme Court Concludes State Universities May Revoke Degrees By Former Students For Academic Dishonesty

Click to read the full story

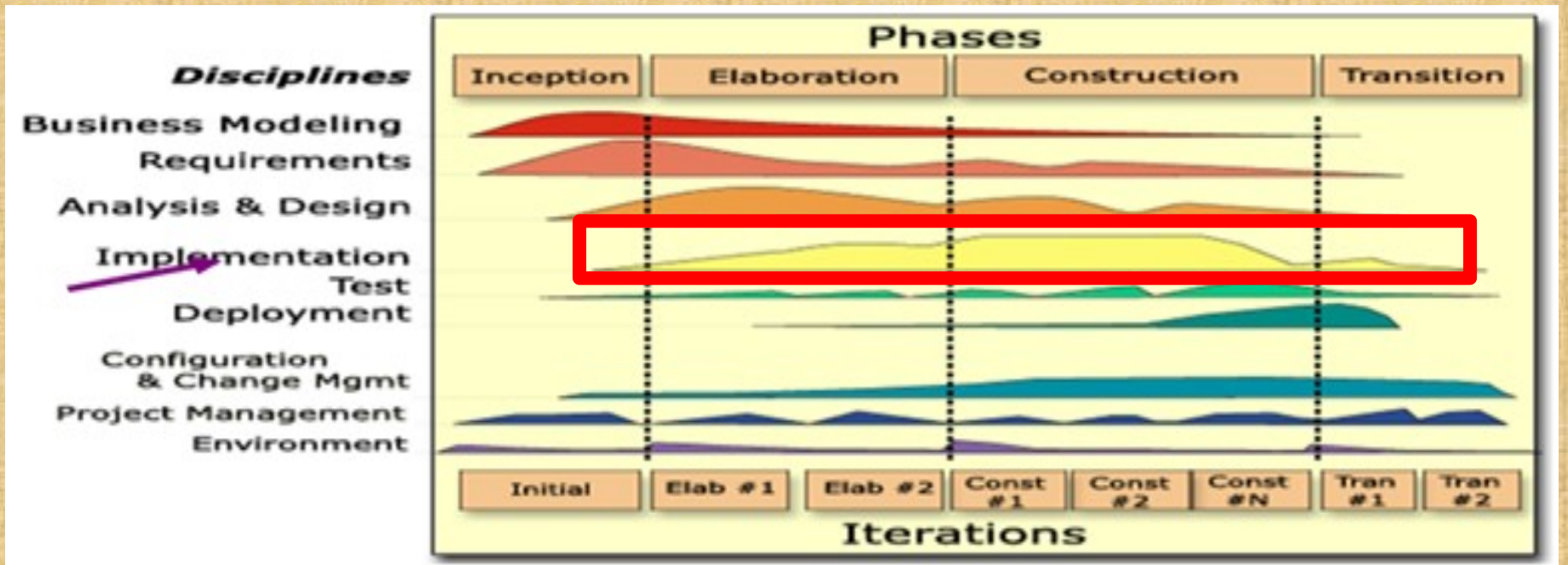
https://www.yahoo.com/news/texas-supreme-court-rules-universities-121510139.html?fr=sycsrp_catchall

From 4:10 to 5:00 – 50 minutes.

11.13.2023
(M 4 to 5:30)

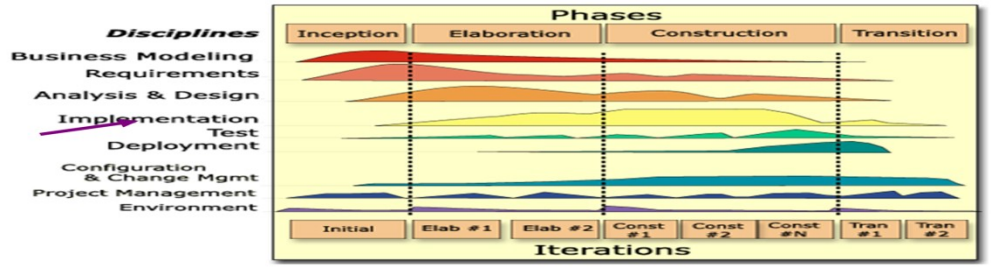
(24)

Lecture 8:
Implementation



Overview

- **Choice** of programming language
- **Fourth** Generation Languages - **4GL**
- **Good programming practices**
- Coding standards
- Code **reuse**
- **Integration**
- **CASE Tools for Implementation Workflow**
- **Metrics** for the **Implementation Workflow**
- **Challenges** of the **Implementation Workflow**

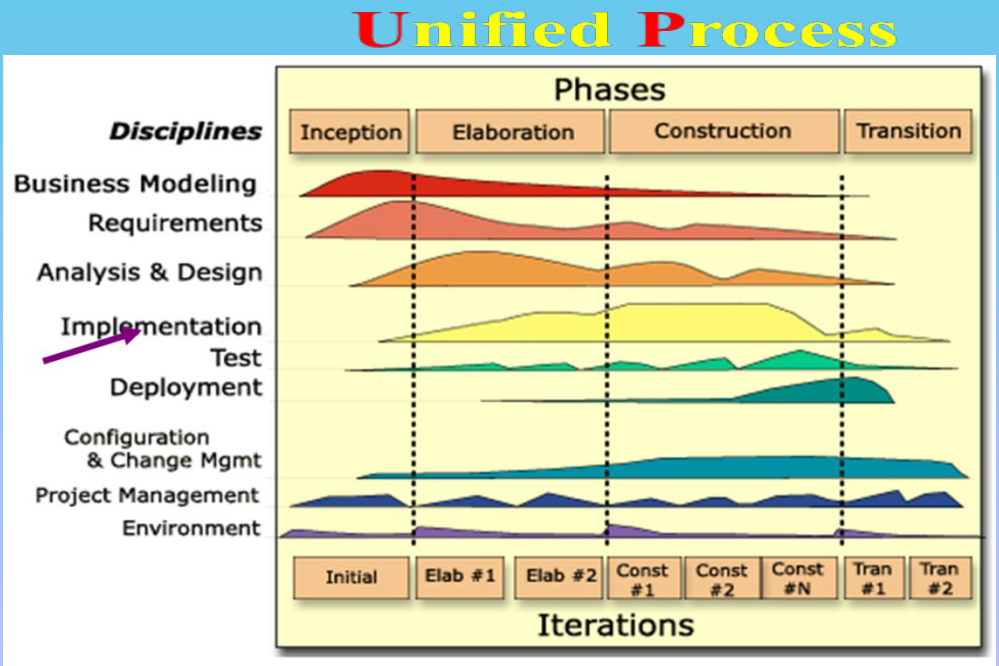


Implementation Workflow

- Real-life **products** are generally too large to be implemented by **a single programmer**

having the "blue print" means 80% done "house"/coding

- This lecture therefore deals with **programming-in-the-many**



10.1 Programming languages

Programming languages

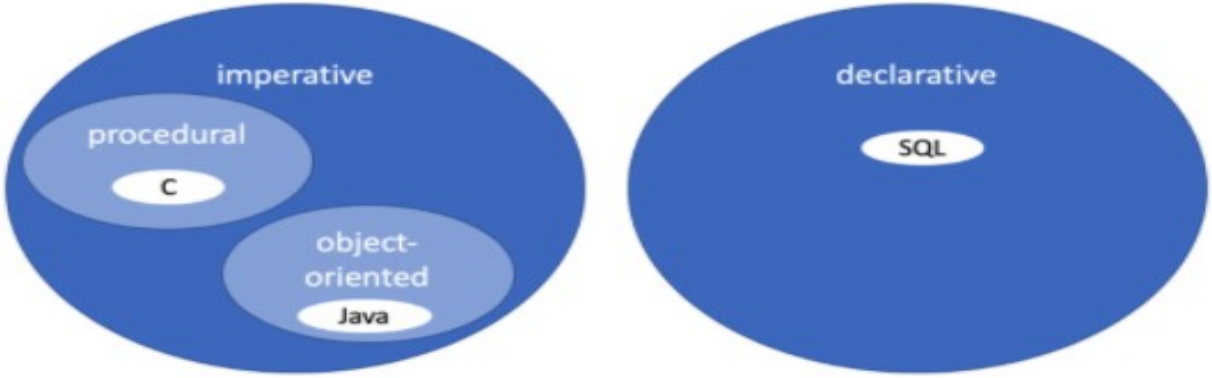
Programming languages fall into two broad categories, or paradigms: imperative and declarative.

Imperative languages contain control flow statements that determine the execution order of program steps. Control flow statements include loops for repeatedly executing code and conditionals for conditionally executing code. Two popular types of imperative languages are:

1. ***Procedural languages*** are composed of procedures, also called functions or subroutines. Most languages developed prior to 1990 are procedural. Ex: C and COBOL.
2. ***Object-oriented languages*** organize code into classes. A class combines variables and procedures into a single construct. Most languages developed since 1990 are object-oriented. Ex: Java, Python, and C++.

Declarative languages do not contain control flow statements. Each statement declares *what* result is desired, using logical expressions, rather than *how* the result is processed. Compilers for declarative languages are called ***optimizers***, since the compiler determines an optimal way to process each declarative statement. SQL is the leading example of a declarative language.

Figure 10.1.1: Programming languages.



Choice of Programming Language

The **Language** is usually specified in the **contract (SRS)**

But what if the **contract (SRS)** specifies that

- The **product** is to be **implemented** in the "**most suitable**" programming **Language**

What **Language** should be chosen?

Choice of Programming **Language**

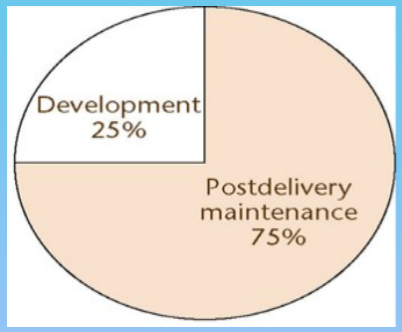
Example

- QQQ Corporation has been writing **COBOL** programs for over 25 years
- Over 200 software **staff**, all with **COBOL** expertise
- What is "**the most suitable**" programming **Language**?

Obviously **COBOL**

Choice of Programming Language

- What happens when new language (C++) is introduced?
  - C++ professionals must be hired
  - Existing COBOL professionals must be retrained
  - Future products are written in C++
  - Existing COBOL products must be maintained
  - There are two classes of programmers
    - COBOL maintainers (despised)
    - C++ developers (paid more)
  - Expensive software, and the hardware to run it, are needed
  - 100s of person-years of expertise with COBOL are wasted



Choice of Programming **Language**

The only possible conclusion

- **COBOL** is the "most suitable" Programming **Language**

And yet, the "most suitable" **Language** for the latest **project** *may* be **C++**

- **COBOL** is suitable for only **data processing applications**

How to choose a Programming **Language**

- **Cost–Benefit Analysis**
  - » Compute **Costs** and **Benefits** of all relevant **Languages**

EXAM 4 REVIEW

Choice of Programming **Language**

Which is the most appropriate **O**bject **O**riented **L**anguage?

- **C++** is not encapsulating main in a Class (**no BIG DEAL!**)
- **Java** enforces the **O**bject **O**riented **P**aradigm (**???**)
- **Training** in the **O**bject **O**riented **P**aradigm is essential before adopting any **O**bject **O**riented **Language**

UML Modeling

JAVA, C++, C#, PYTHON, RUBY, PHP

What about choosing a Fourth Generation **Language** (4GL)?

Fourth Generation Languages

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

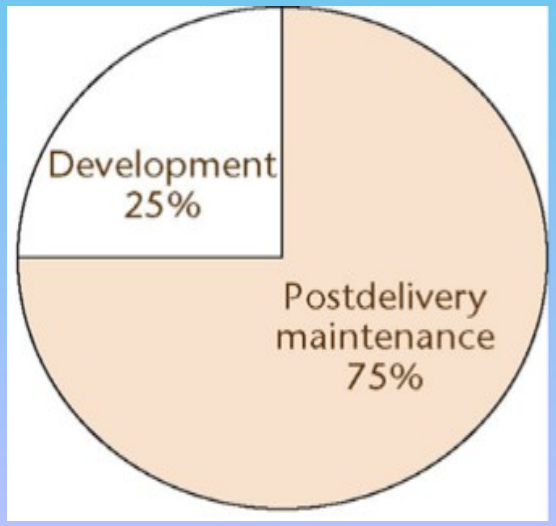
Fourth Generation Languages

- Fourth generation languages (4GLs)
  - One **3GL** statement is equivalent to 5–10 assembly statements
  - Each **4GL** statement was intended to be equivalent to 30 or even 50 assembly statements

Fourth Generation Languages

It was hoped that 4GLs would

- Result in **applications** that are easy to **build** and quick to **change**
  - » Reducing **maintenance** costs
- Simplify debugging
- Make languages **user** friendly
  - » Leading to **end-user** **programming**



Fourth Generation Languages

The **power** of a nonprocedural **Language**, and the **price**

RoR, **SQL**

Productivity Increases with a 4GL?

The picture is not uniformly rosy

Playtex used **ADF**, obtained an **80** to **1 productivity increase over COBOL**

- However, **Playtex** then **used COBOL** for later applications

ADF integrates a mix of subframeworks to provide the key functions for object-relational mapping and other forms of service access, data bindings, and user interface, along with the functional glue to hold it all together. ADF stands for 'Applications Development Framework' and it's developed by Oracle. ADF is a framework and it'll help you build your applications easily as many of the redundant things that we do while we develop our applications are taken care by the framework.

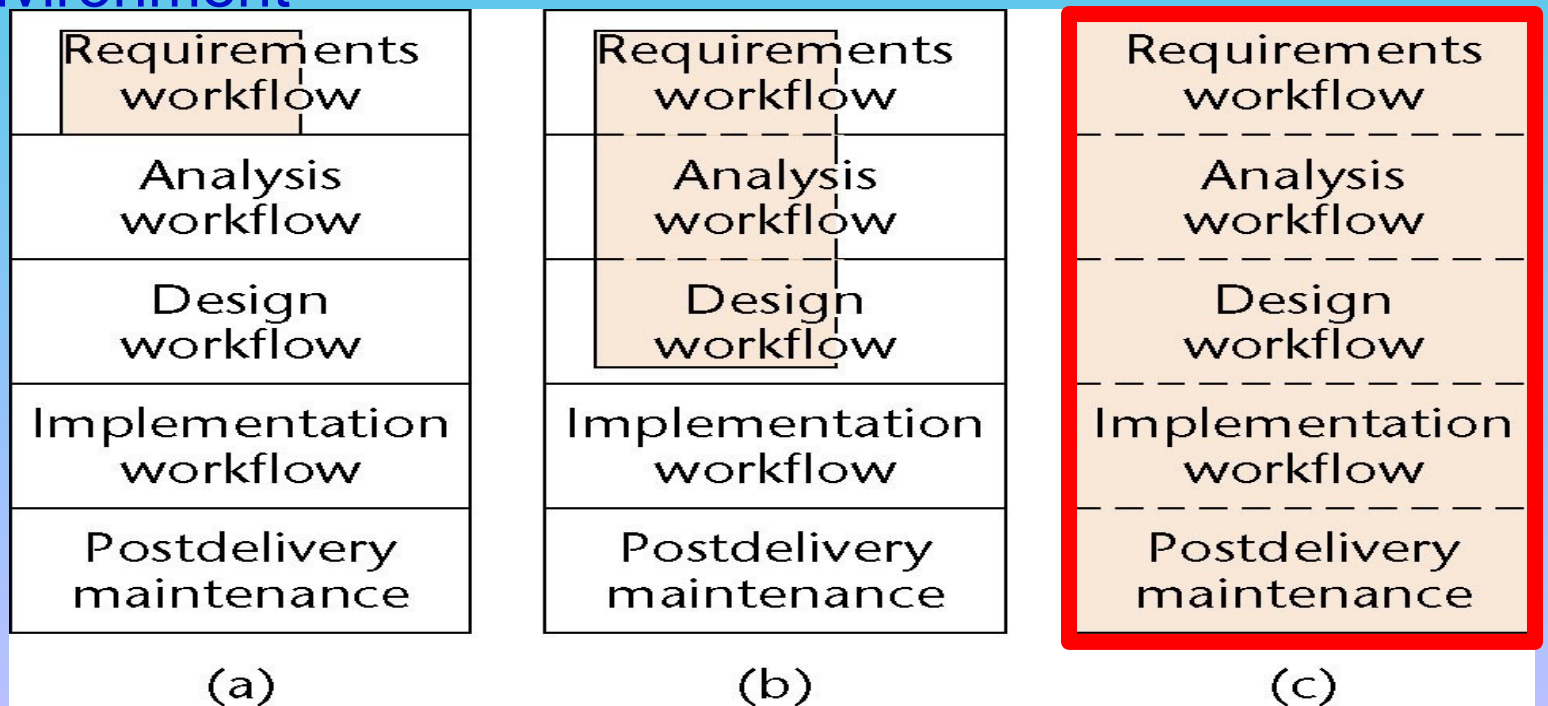
4GL productivity increases of **10** to **1** over **COBOL** have been reported

However, there are **plenty of reports of bad experiences**

Actual Experiences with 4GLs

Many **4GL**s are supported by powerful CASE environments

- **This is a problem for organizations at CMM level 1 or 2**
- Some reported **4GL failures** are due to the underlying CASE environment



Actual Experiences with 4GLs

- Attitudes of **43** organizations to 4GLs
  - Use of 4GL **reduced users' frustrations**
  - 4GLs are **slow** and **inefficient**, on average
  - Overall, **28** organizations using 4GL for over **3** years felt that the **benefits outweighed the costs**

Dangers of a 4GL

End-user programming

- Programmers are taught to mistrust computer output
- **End users** are taught to believe computer output
- An **end-user** updating a **database** can be particularly dangerous

Dangers of a 4GL

- Potential pitfalls for **management**
  - **Premature** introduction of a CASE environment
  - Providing **insufficient training** for the development team
  - Choosing the **wrong** 4GL

22 software development trends for 2022

https://blog.codacy.com/software-development-trends-2022/

Believe it or not, the year 2022 is right around the corner! So what does next year have in store for the rapidly changing, ever-evolving software world? From code reviews to DevOps, software testing, and tech companies' culture, here are our 22 software development trends for 2022 🔮

#1 – A rise of automated code reviews

Tech companies increasingly see a well-defined code review process as a fundamental part of the software development process. Code reviews are among the best ways to improve code quality. Automated code review tools are increasing in popularity as more companies start to include them in their code review process, allowing developers to spend more time building new features instead of on code reviews. We can expect solutions like Codacy to see an increase in adoption in 2022.

Believe it or not, the year 2022 is right around the corner! So what does next year have in store for the rapidly changing, ever-evolving software world? From code reviews to DevOps, software testing, and tech companies' culture, here are our 22 software development trends for 2022 🔮

#2 – A greater focus on software quality standards

Software solutions are increasingly embedded in our day-to-day lives and in most of the devices we use. As a result, there is a growing need for software to follow quality standards like the ones proposed by ISO. Examples include the ISO/IEC 25010 on software product quality or the ISO/IEC 27001 on information technology. Companies are also starting to see the advantages of ISO certification like improved quality, more efficient processes, increased reputation, and enhanced client satisfaction.

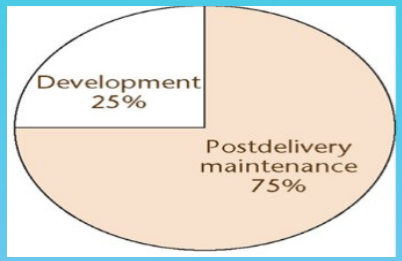
Believe it or not, the year 2022 is right around the corner! So what does next year have in store for the rapidly changing, ever-evolving software world? From code reviews to DevOps, software testing, and tech companies' culture, here are our 22 software development trends for 2022 🔮

#3 – A pragmatic focus on coding standards

As companies and teams grow, they start to have a list of rules and guidelines for writing code. This list also incorporates language conventions and style consistency. Having a clear standard can help current developers, and it is also relevant for the onboarding of new developers. In 2022, we expect more companies to see the advantages of coding standards, not motivated by growing pains but by understanding its need from the early stages.

Good Programming Practices

- Use of ***Consistent*** **and** ***Meaningful*** **variable names**
  - "**Consistent**" to aid future **Maintenance Programmers**



  - "**Meaningful**" to aid future **Maintenance Programmers**

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
   float expectedWeeklyInvestmentReturn;                    (expected weekly investment return)
   float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                            (expected total mortgage payments
                                                            less total weekly grants)
   float estimatedFunds = (float) 0.0;                      (total estimated funds for week)
```

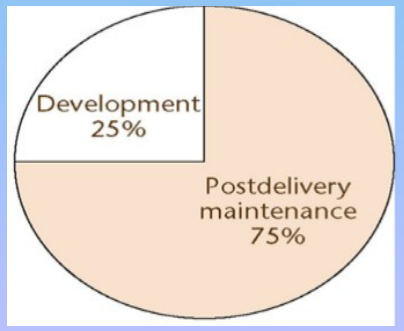
Use of **Consistent** and **Meaningful Variable Names**

- A **code artifact** includes the **variable names** `freq`**`Average`**, `frequency`**`Maximum`**, **`min`**`fr`, `frqncy`**`Totl`**

- A **Maintenance Programmer** has to know if `freq`, `frequency`, `fr`, `frqncy` all refer to the **same** thing

  - **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



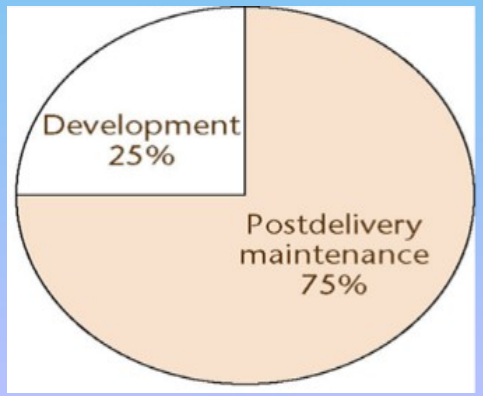
Consistent and Meaningful Variable Names

We **can use** `frequencyAverage`, `frequencyMaximum`, `frequencyMinimum`, `frequencyTotal`

lower Camel notation

We **can also use** `averageFrequency`, `maximumFrequency`, `minimumFrequency`, `totalFrequency`

But all four names **must come from the same set**



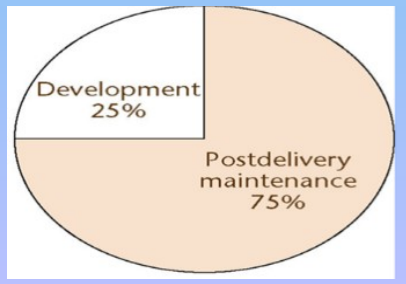
The Issue of Self-Documenting Code

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**

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
  float expectedWeeklyInvestmentReturn;          (expected weekly investment return)
  float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                 (expected total mortgage payments
                                                 less total weekly grants)
  float estimatedFunds = (float) 0.0;            (total estimated funds for week)
```



Self-Documenting Code Example

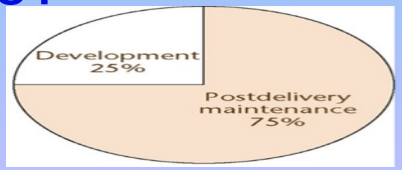
Example:

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
  float expectedWeeklyInvestmentReturn;          (expected weekly investment return
  float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                 (expected total mortgage payments
                                                 less total weekly grants)
  float estimatedFunds = (float) 0.0;            (total estimated funds for week)
```

- **Code artifact** contains the **variable**

```
xCoordinateOfPositionOfRobotArm
```

- This is abbreviated to **`xCoord`**
- **This is fine**, because the entire **module** deals with the movement of the robot arm
- But does the **Maintenance Programmer** know this?



Prologue Comments

- Minimal **Prologue 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

SQAs review code!

Other Comments

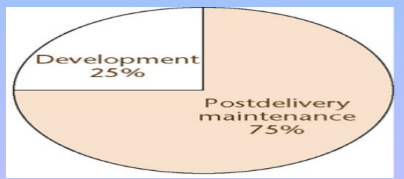
Suggestion

- **Comments** are essential ONLY whenever the **code** is written in a non-obvious way, or **makes use** of some subtle aspect of the **Language**

```
public static void computeEstimatedFunds( )

This method computes the estimated funds available for the week.

{

  float expectedWeeklyInvestmentReturn;          (expected weekly investment return)

  float expectedTotalWeeklyNetPayments = (float) 0.0;

                                                 (expected total mortgage payments
                                                 less total weekly grants)

  float estimatedFunds = (float) 0.0;            (total estimated funds for week)
```

Nonsense!

- **ReCode** in a clearer way
- We **must never promote/excuse poor Programming**
- However, **comments can assist future Maintenance Programmers**



Use of Parameters

- 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

Code Layout for Increased Readability

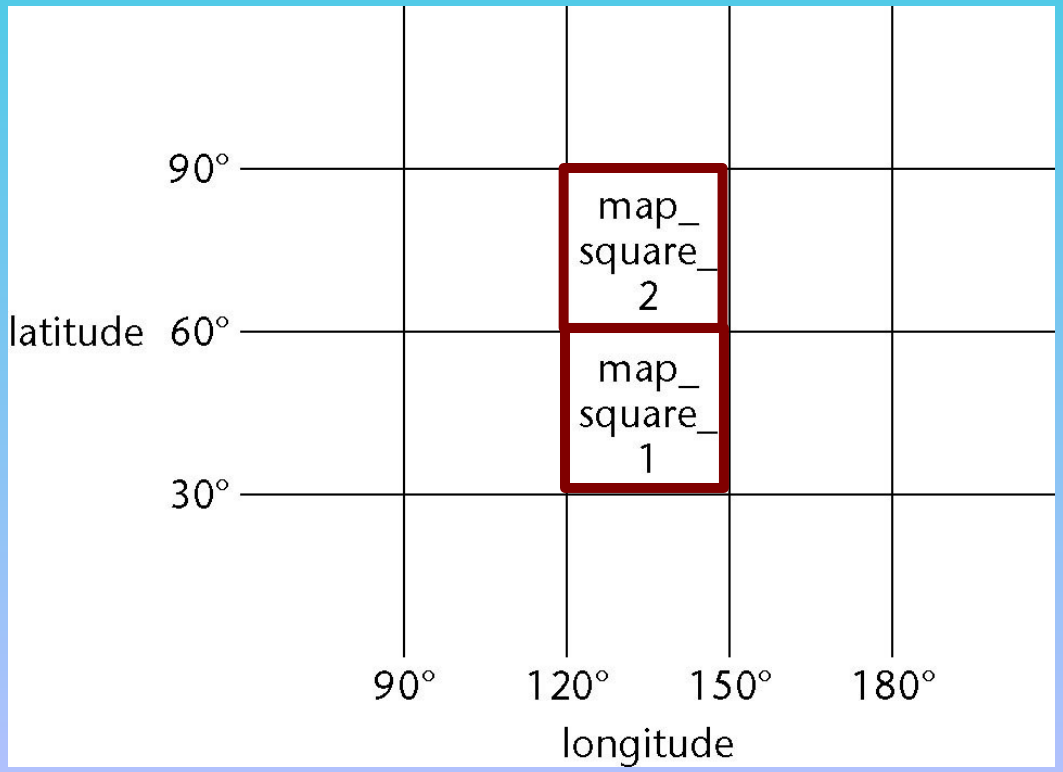
- Use indentation
- Better, use a pretty-printer
- Use plenty of blank lines
  - To break up big blocks of code

IDEs do that for you automatically!

Nested `if` Statements

Example

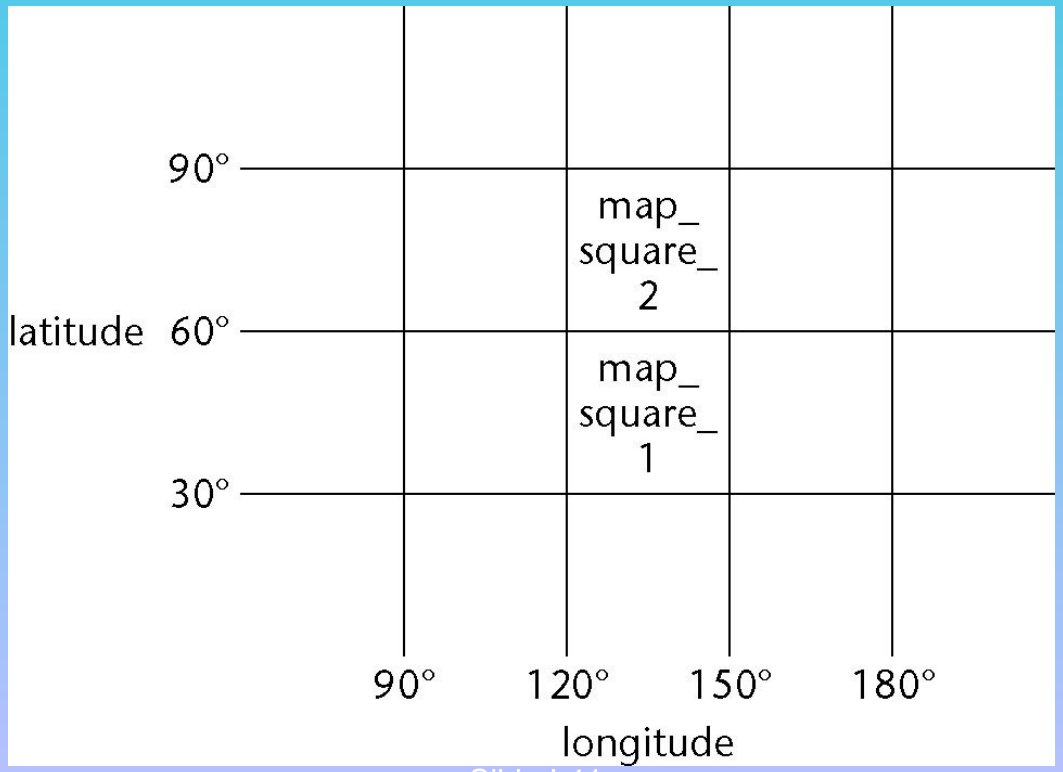
- A map consists of **two squares**. Write **code** to determine whether a point on the Earth's surface lies in `map_square_1` or `map_square_2`, or **is not on the map**



Nested `if` Statements

- Solution 1. **Badly formatted**

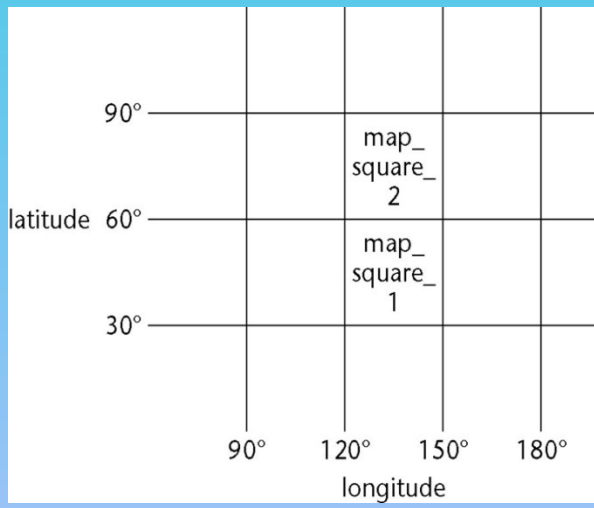
```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```



Nested `if` Statements

- Solution 2. Well-formatted, **badly constructed**

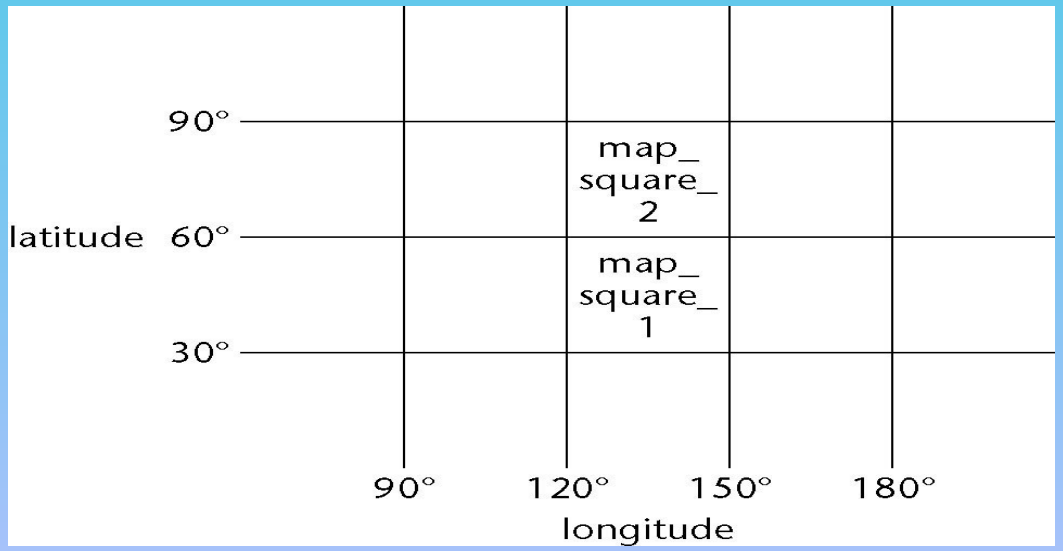
```
if (latitude > 30 && longitude > 120)
{
  if (latitude <= 60 && longitude <= 150)
    mapSquareNo = 1;
  else
    if (latitude <= 90 && longitude <= 150)
      mapSquareNo = 2;
    else
      print "Not on the map";
}
else
  print "Not on the map";
```



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";
```



EXAM 4 REVIEW

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>
```

Nested `if` Statements

- Rule of thumb
  - `if` statements nested to a **depth of greater than three** should be avoided as **poor Programming Practice**

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"

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

- Better
  - "**Programmers should consult their managers** before constructing a **module** with fewer than 35 or more than 50 executable statements"

Remarks on Programming **Standards**

- No **Standard** can ever be universally applicable
- **Standards** imposed from above **will be ignored**

UML

- **Standard** must be **checkable** by machine

UML "compilable"

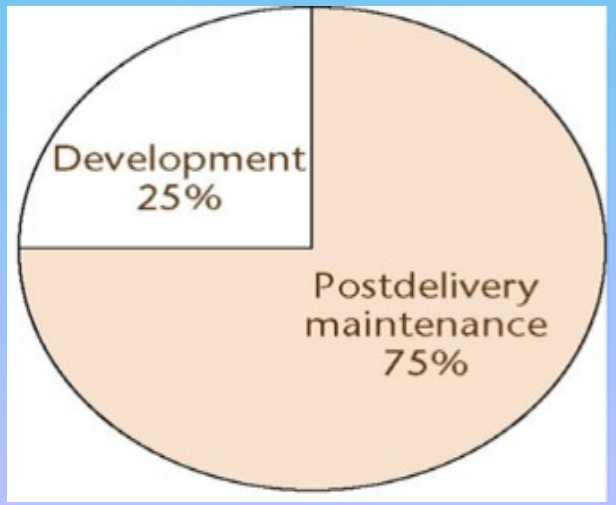
Examples of **Good Programming Standards**

- "Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the **T**eam **L**eader"
- "**modules** should consist of between 35 and 50 statements, except with prior approval from the **T**eam **L**eader"
- "Use of `goto`s should be avoided. However, with prior approval from the **T**eam **L**eader, a forward `goto` may be used for error handling" (**use TRY CATCH** BLOCKS!)

Remarks on Programming **Standards**

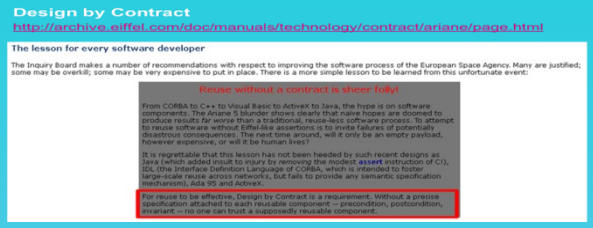
The aim of **Standards** is to make **Maintenance** **easier**

- If they make **Development** difficult, then they **must be modified**
- Overly restrictive **Standards** are **counterproductive**
- The **quality** of **Software** suffers



Code Reuse

Code Reuse is the most common form of **Reuse**


Design by Contract
http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

The lesson for every software developer

The Inquiry Board makes a number of recommendations with respect to improving the software process of the European Space Agency. Many are justified; some may be overkill; some may be very expensive to put in place. There is a more simple lesson to be learned from this unfortunate event:

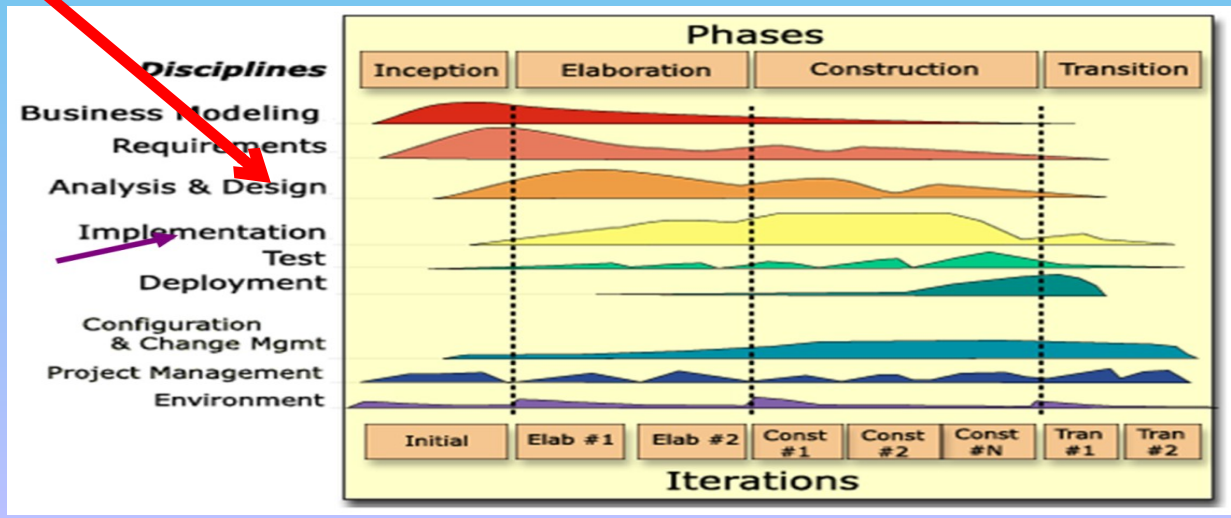
Reuse without a contract is sheer folly!

From CORBA to C++ to Visual Basic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naive hopes are doomed to produce results far worse than a traditional, reuse-less software process. To attempt to reuse software without Eiffel-like assertions is to invite failures of potentially disastrous consequences. The next time around, will it only be an empty payload, however expensive, or will it be human lives?

It is regrettable that this lesson has not been heeded by such recent designs as Java (which added insult to injury by removing the modest **assert** instruction of C), IDL (the Interface Definition Language of CORBA, which is intended to foster large-scale reuse across networks, but fails to provide any semantic specification mechanism), Ada 95 and ActiveX.

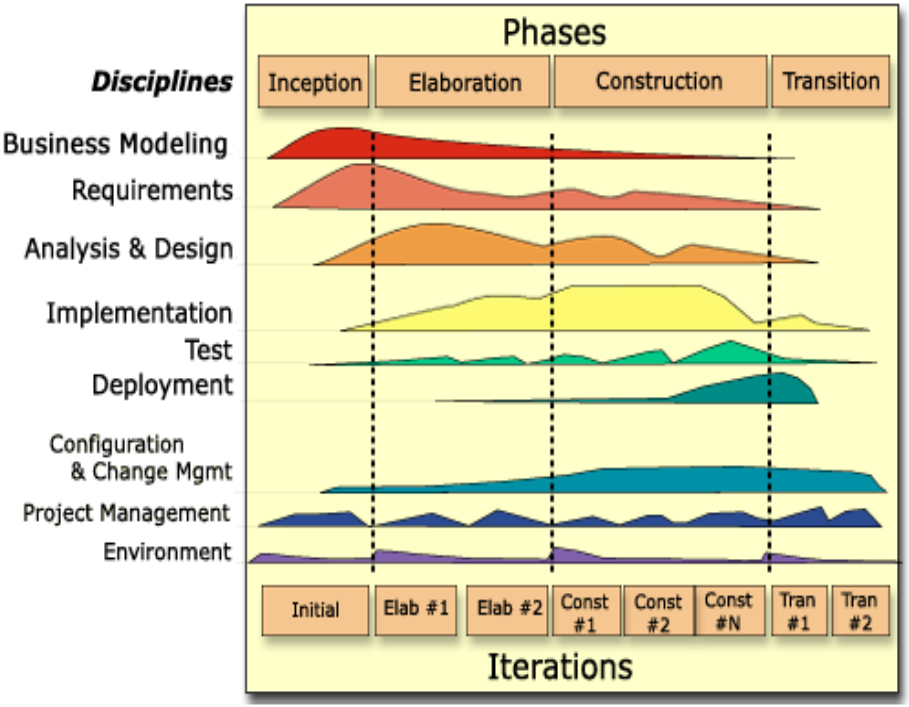
For reuse to be effective, Design by Contract is a requirement. Without a precise specification attached to each reusable component -- precondition, postcondition, invariant -- no one can trust a supposedly reusable component.

However, **artifacts from all Workflows** can be **Reused** (**Design Patterns**)



EXAM 4 REVIEW

https://se.inf.ethz.ch/~meyer/publications/computer/ariane.pdf



A recent $500 million software error provides a sobering reminder that this principle is not just a pleasant academic ideal. On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed, about 40 seconds after takeoff. Media reports indicated that a half-billion dollars was lost—the rocket was uninsured.

The French space agency, CNES (Centre National d'Etudes Spatiales), and the European Space Agency immediately appointed an international inquiry board, made up of respected experts from major European countries, which produced a report in hardly more than a month. These agencies are to be commended for the speed and openness with which they handled the disaster. The report is available on the Web, in both French and English (http://www.cnes.fr/actualites/news/rapport_501.html).

It is a remarkable document: short, clear, and forceful. The explosion, the report says, is the result of a software error, possibly the costliest in history (at least in dollar terms, since earlier cases have cost lives).

Particularly vexing is the realization that the error came from a piece of the software that was *not* needed. The software involved is part of the Inertial Reference System, for which we will keep the acronym SRI used in the report, if only to avoid the unpleasant connotation that the reverse acronym has for US readers. Before liftoff, certain computations are performed to align the SRI. Normally, these computations should cease at −9 seconds, but because there is a chance that a countdown could be put on hold, the engineers gave themselves some leeway. They reasoned that, because resetting the SRI could take

https://se.inf.ethz.ch/~meyer/publications/computer/ariane.pdf

several hours (at least in earlier versions of Ariane), it was better to let the computation proceed than to stop it and then have to restart it if liftoff was delayed. So the SRI computation continues for 50 seconds after the start of flight mode—well into the flight period. After takeoff, of course, this computation is useless. In the Ariane 5 flight, however, it caused an exception, which was not caught and—boom.

The exception was due to a floating-point error during a conversion from a 64-bit floating-point value, representing the flight's "horizontal bias," to a 16-bit signed integer: In other words, the value that was converted was greater than what can be represented as a 16-bit signed integer. There was no explicit exception handler to catch the exception, so it followed the usual fate of uncaught exceptions and crashed the entire software, hence the onboard computers, hence the mission.

How in the world could such a trivial error have remained undetected and cause a $500 million rocket to blow up?

YOU CAN'T BLAME MANAGEMENT

Although something clearly went wrong in the validation and verification process (or we wouldn't have a story to tell), and although the Inquiry Board does make several recommendations to improve the process, it is also clear that systematic documentation, validation, and management procedures were in place.

The software engineering literature has often contended that most software problems are primarily management problems. This is not the case here: the problem was a technical one. (Of course you can always argue that good management will spot technical problems early enough.)

YOU CAN'T BLAME THE LANGUAGE

Ada's exception mechanism has been criticized in the literature, but in this case it could have been used to catch the exception. In fact, the report says:

> *Not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer. To determine the vulnerability of unprotected code, an analysis was performed on every operation which could give rise to an ... operand error. This led to protection being added to four of [seven] variables ... in the Ada code. However, three of the variables were left unprotected.*

YOU CAN'T BLAME THE DESIGN

Why was the exception not monitored? The analysis revealed that overflow (a horizontal bias not fitting in a 16-bit integer) could not occur. Was the analysis wrong? No! It was right *for the Ariane 4 trajectory*. For Ariane 5, with other trajectory parameters, it did not hold.

YOU CAN'T BLAME THE IMPLEMENTATION

Some may criticize removing the conversion protection to achieve more performance (the 80 percent workload target), but this decision was justified by the theoretical analysis. To engineer is to make compromises. If you have proved that a condition cannot happen, you are entitled not to check for it. If every program checked for all possible and impossible events, no useful instruction would ever get executed!

YOU CAN'T BLAME TESTING

The Inquiry Board recommends better testing procedures, and it also recommends testing the entire system rather than parts of it (in the Ariane 5 case the SRI and the flight software were tested separately). But even if you can test more, you can never test all. Testing, as we all know, can show the presence of errors, not their absence. The only fully realistic test is a launch. And in fact, the launch was a test launch, in that it carried no commercial payload, although it was probably not intended to be a $500 million test.

YOU CAN TRY TO BLAME REUSE

The SRI horizontal bias module was indeed reused from 10-year-old software, the software from Ariane 4. But this is not the real story.

BUT YOU REALLY HAVE TO BLAME REUSE *SPECIFICATION*

What was truly unacceptable in this case was the absence of any kind of precise specification associated with this reusable module. The requirement that the horizontal bias should fit on 16 bits was in fact stated in an obscure part of a mission document. But it was nowhere to be found in the code itself!

One of the principles of design by contract, as earlier columns have said, is that *any* software element that has such a fundamental constraint should state it explicitly, as part of a mechanism present in the language. In an Eiffel version, for example, it would be stated as

```
convert (horizontal_bias:
 DOUBLE): INTEGER  is
 require
    horizontal_bias
       <= Maximum_bias
 do
    ...
 ensure
    ...
 end
```

where the precondition (`require...`) states clearly and precisely what the input must satisfy to be acceptable.

Does this mean that the crash would automatically have been avoided had the mission used a language and method supporting built-in assertions and design by contract? Although it is always risky to draw such after-the-fact conclusions, the answer is probably yes:

- Assertions (preconditions and postconditions in particular) can be automatically turned on during testing, through a simple compiler option. The error might have been caught then.
- Assertions can remain turned on during execution, triggering an exception if violated. Given the performance constraints on such a mission, however, this would probably not have been the case.

Assertions (preconditions and postconditions in particular) can be automatically turned on during testing, through a simple compiler option. The error might have been caught then.
Assertions can remain turned on during execution, triggering an exception if violated. Given the performance constraints on such a mission, however, this would probably not have been the case.

Design by Contract

http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

The lesson for every software developer

The Inquiry Board makes a number of recommendations with respect to improving the software process of the European Space Agency. Many are justified; some may be overkill; some may be very expensive to put in place. There is a more simple lesson to be learned from this unfortunate event:

Reuse without a contract is sheer folly!

From CORBA to C++ to Visual Basic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naive hopes are doomed to produce results *far worse* than a traditional, reuse-less software process. To attempt to reuse software without Eiffel-like assertions is to invite failures of potentially disastrous consequences. The next time around, will it only be an empty payload, however expensive, or will it be human lives?

It is regrettable that this lesson has not been heeded by such recent designs as Java (which added insult to injury by *removing* the modest assert instruction of C!), IDL (the Interface Definition Language of CORBA, which is intended to foster large-scale reuse across networks, but fails to provide any semantic specification mechanism), Ada 95 and ActiveX.

For reuse to be effective, Design by Contract is a requirement. Without a precise specification attached to each reusable component -- precondition, postcondition, invariant -- no one can trust a supposedly reusable component.

Integration

- The approach up to now:
  - **Implementation** followed by **Integration**
- This is a **poor approach**
- Better:
  - Combine **Implementation** and **Integration** methodically

Product with 13 modules

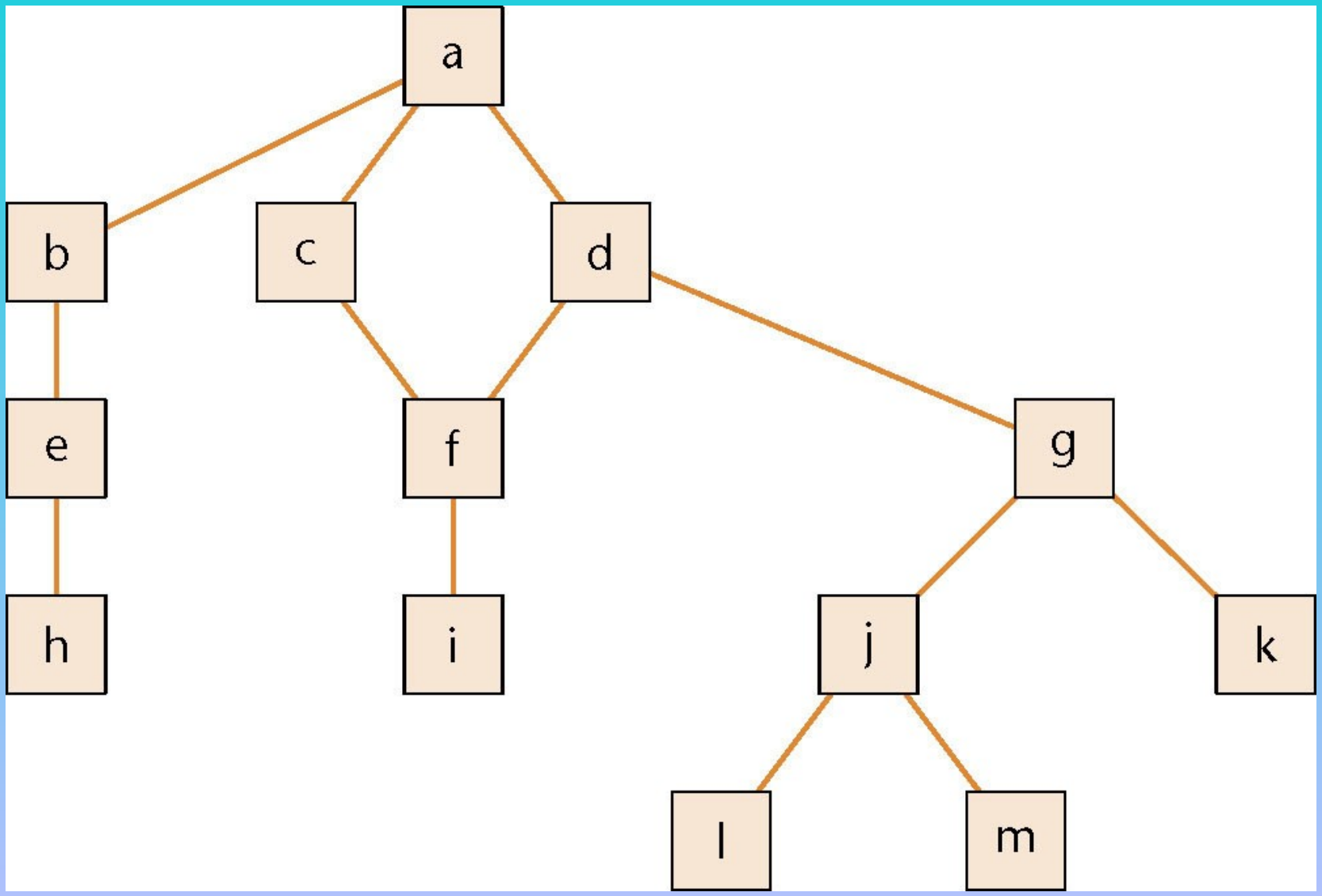
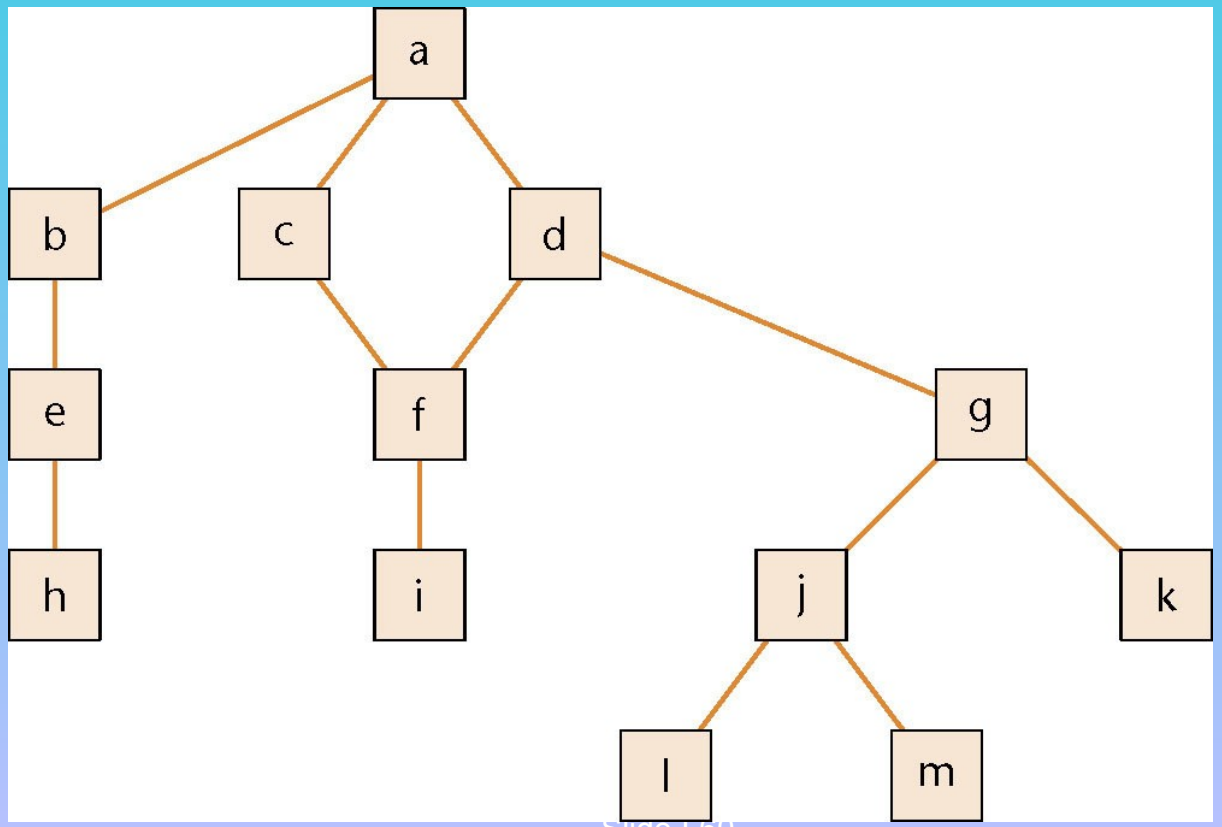


Figure 14.6

Implementation, then **Integration**

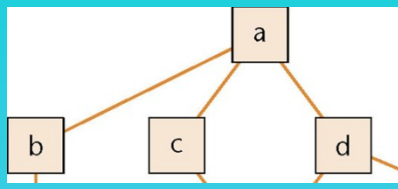
Code and **Test** each **Code** artifact separately

Link all **13** artifacts together, **test** the **product** as a whole



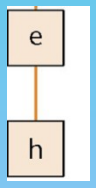
Drivers and Stubs

To test artifact `a`, artifacts `b,c,d` must be **stubs**



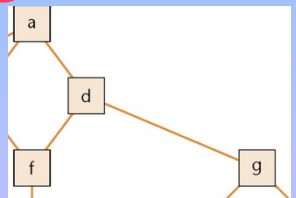
- An empty artifact, or
- Prints a message ("`Procedure radarCalc called`"), or
- Returns precooked values from **preplanned test cases**

To test artifact `h` on its own requires **a driver**, which calls it

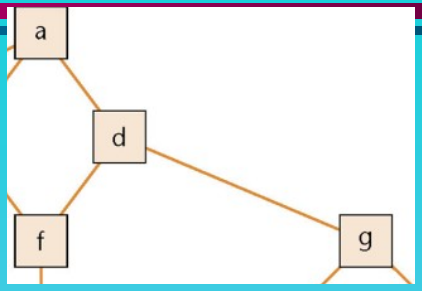


- Once, or
- Several times, or
- **Many times**, each time **checking the value returned**

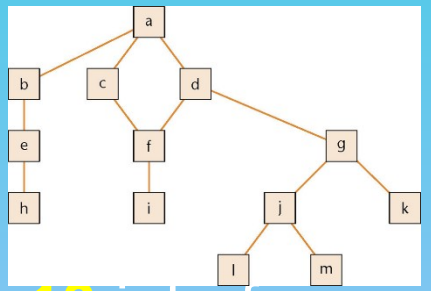
Testing artifact `d` requires **a driver and two stubs**



Implementation, then **Integration**



- Problem 1
  - **stubs** and **drivers** must be written, then **thrown away** after unit testing is complete



- Problem 2
  - Lack of **fault isolation**
  - A **fault** could lie in *any* of the **13** artifacts or **13** interfaces
  - In a **large product** with, say, **103** artifacts and **108** interfaces, there are **211** places where a **fault** might lie

Top-Down Integration

- If **code** artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is **Implemented and Integrated before** `mBelow`

- One possible Top-Down ordering is
  - `a, b,c,d, e,f,g, h,i,j,k, l,m`

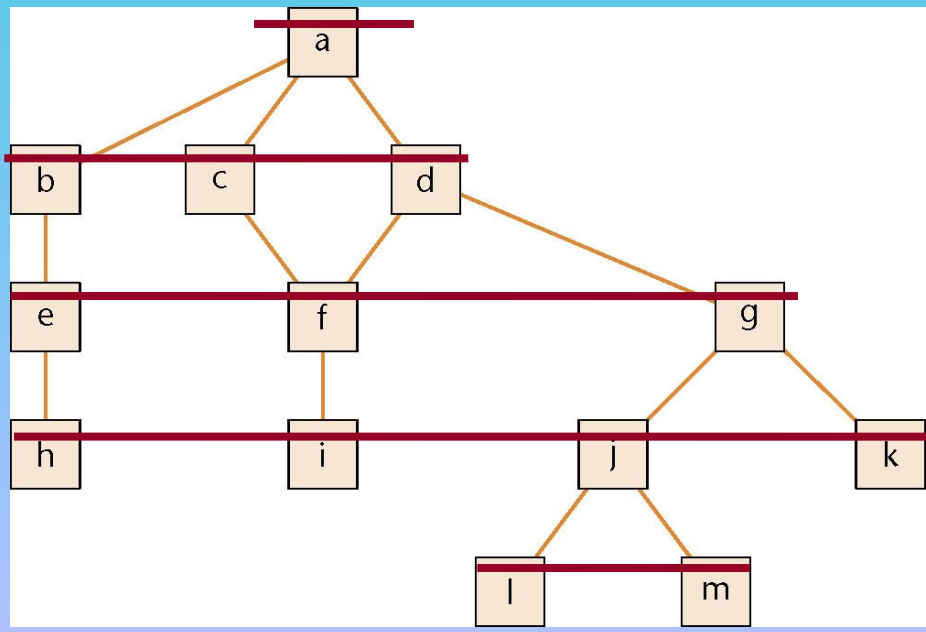
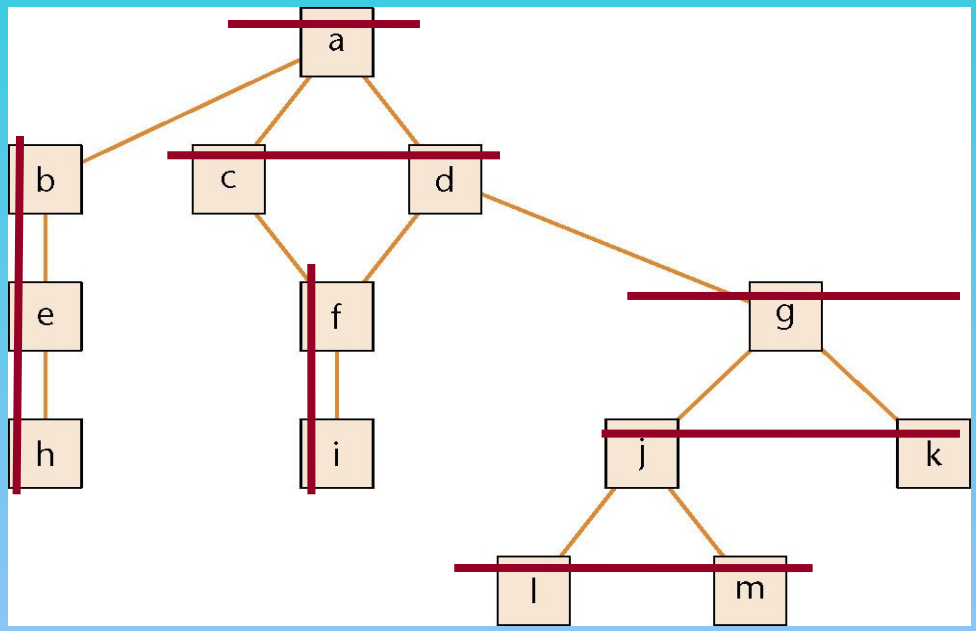


Figure 14.6 (again)

Top-Down **Integration**

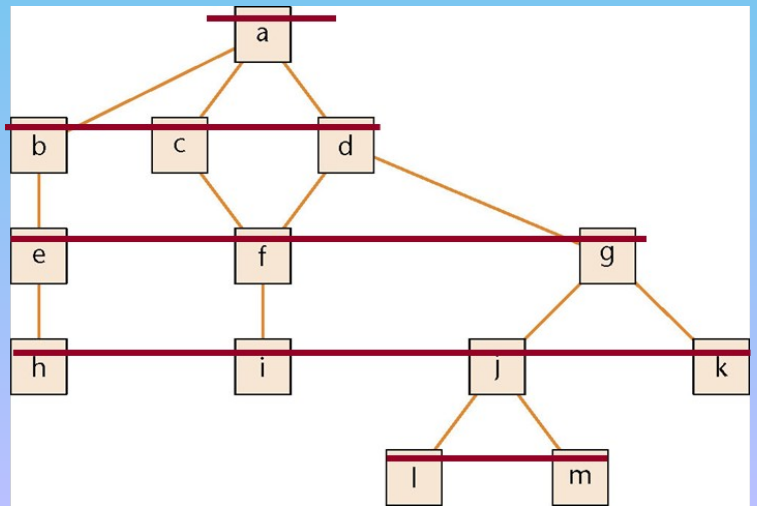
- Another possible Top-Down ordering is

```
          a
[a]       b,e,h
[a]       c,d,f,i
[a,d]     g,j,k,l,m
```



Top-Down Integration

- Advantage 1: **Fault** isolation
  - A previously successful **test case fails** when `mNew` is added to what has been tested so far
    - The **fault** must lie in `mNew` **or** the interface(s) between `mNew` and the **rest of the product**
- Advantage 2: **stubs** are not wasted
  - Each stub is expanded into the corresponding complete artifact at the appropriate step



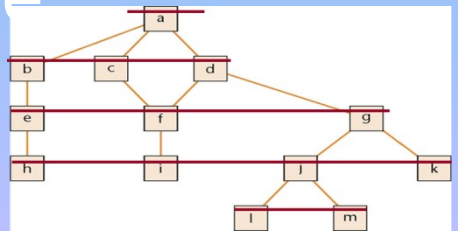
Top-Down **Integration**

- Advantage 3: Major **design flaws** show up early

- Logic artifacts include the decision-making flow of control
  - In the example, artifacts `a,b,c,d,g,j`

- Operational artifacts perform the actual operations of the **product**
  - In the example, artifacts `h,i,k,l,m`

- The Logic artifacts are developed before the Operational artifacts



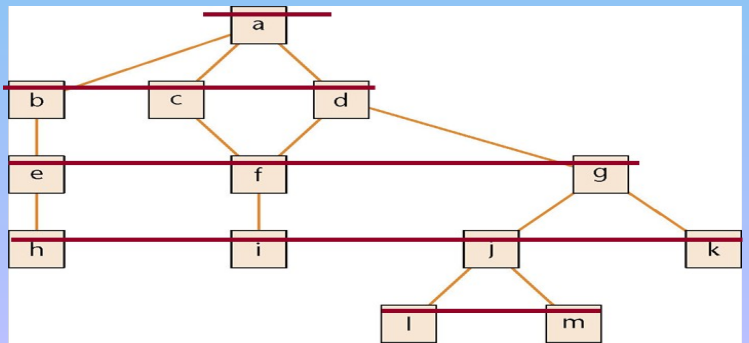
Top-Down **Integration**

- Problem 1
  - **Reus**able artifacts are not properly tested
  - Lower level (Operational) artifacts are not tested frequently

- Defensive programming (fault shielding)
  - Example:

```
if (x >= 0)
y = computeSquareRoot (x, errorFlag);
```

  - `computeSquareRoot` is never tested with `x < 0`
  - This has implications for **Reuse**



Bottom-Up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is **Implemented and Integrated before** `mAbove`

- One possible Bottom-Up ordering is

  `l,m,`
  `h,i,j,k,`
  `e,f,g,`
  `b,c,d,`
  `a`

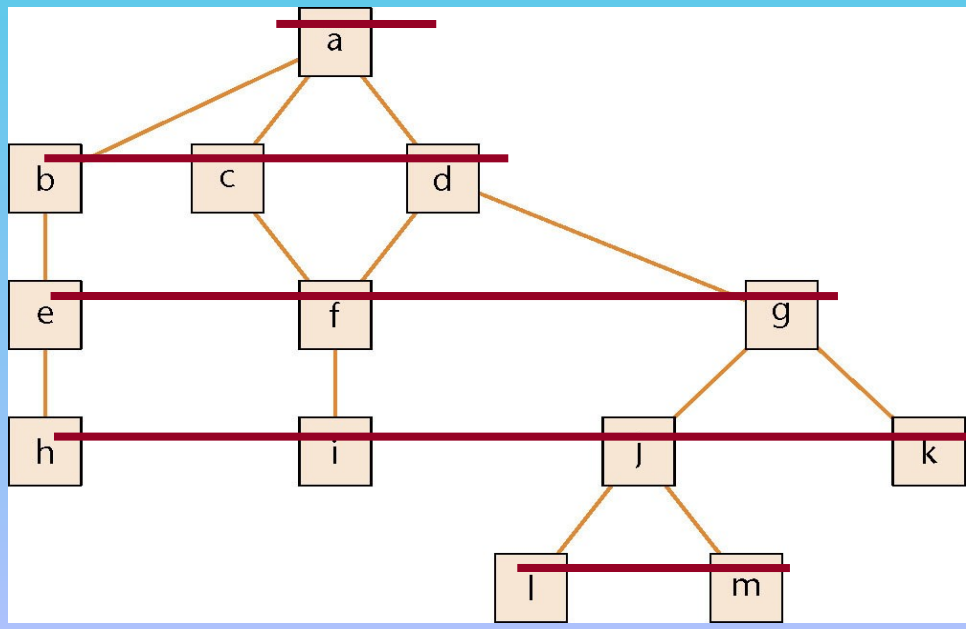
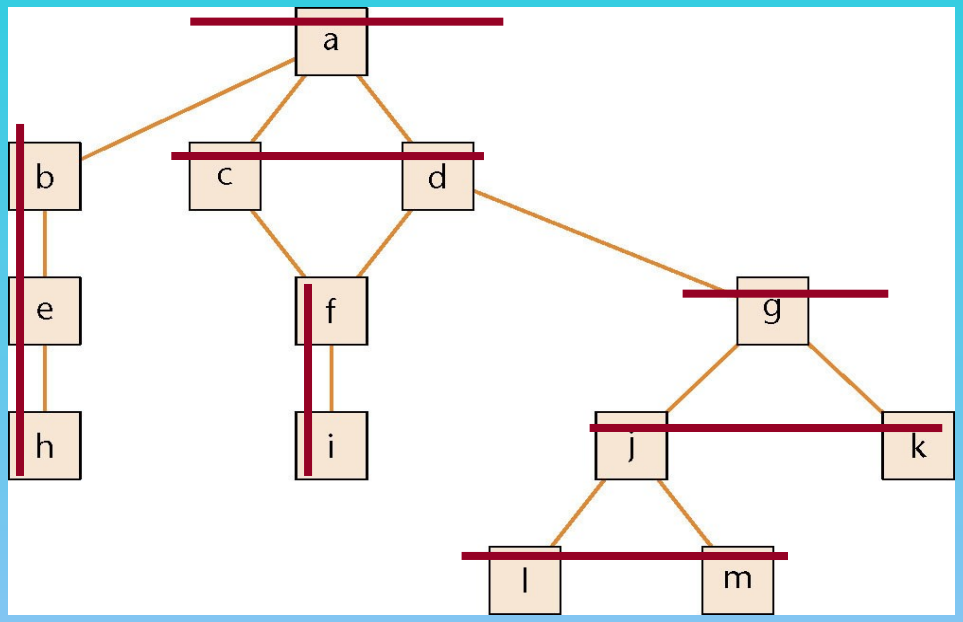


Figure 14.6 (again)

Bottom-Up **Integration**

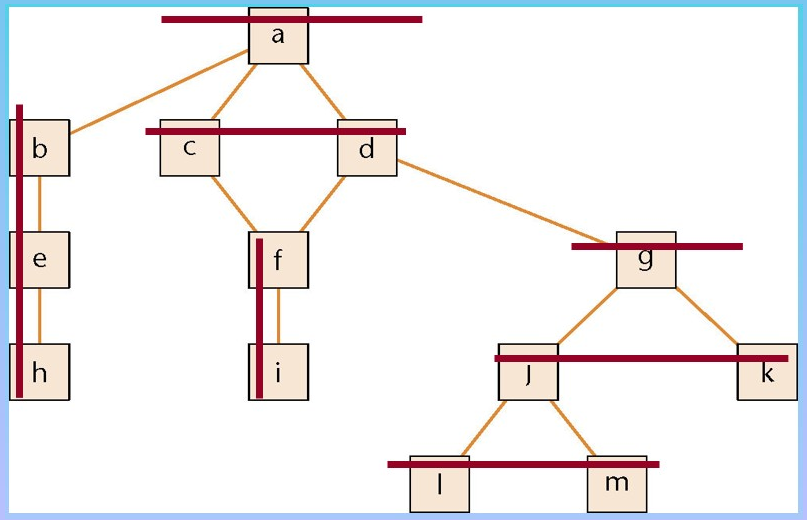
- Another possible Bottom-Up ordering is

```
h,e,b
i,f,c,d
l,m,j,k,g        [d]
a                [b,c,d]
```



Bottom-up **Integration**

- Advantage 1
  - **Operational** artifacts are **thoroughly tested**
- Advantage 2
  - **Operational** artifacts are tested with **drivers**, not by **fault** shielding, defensively programmed artifacts
- Advantage 3
  - **Fault** isolation

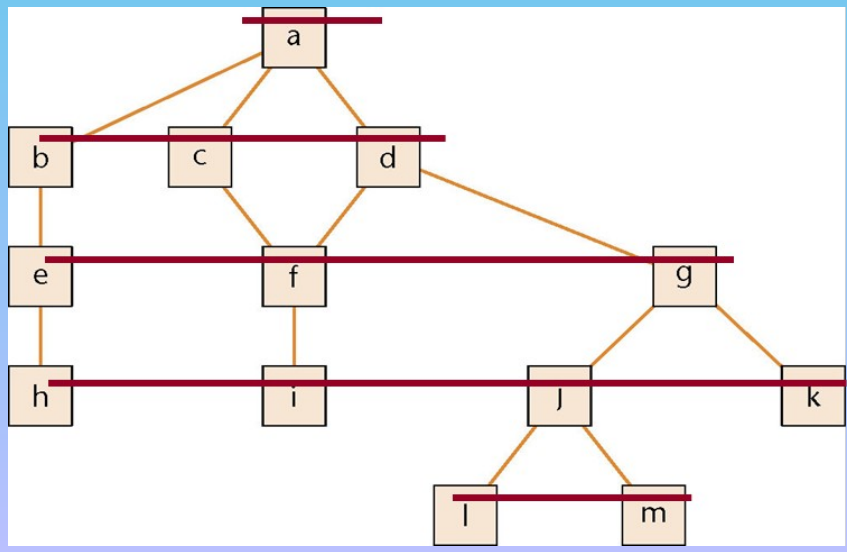


Operational artifacts perform the actual operations of the PRODUCT

- In the example, artifacts h,i,k,l,m

Bottom-up **Integration**

- **Difficulty 1**
  - Major Design **faults** are detected late
- Solution
  - Combine Top-Down and Bottom-Up strategies making use of their **strengths** and minimizing their **weaknesses**

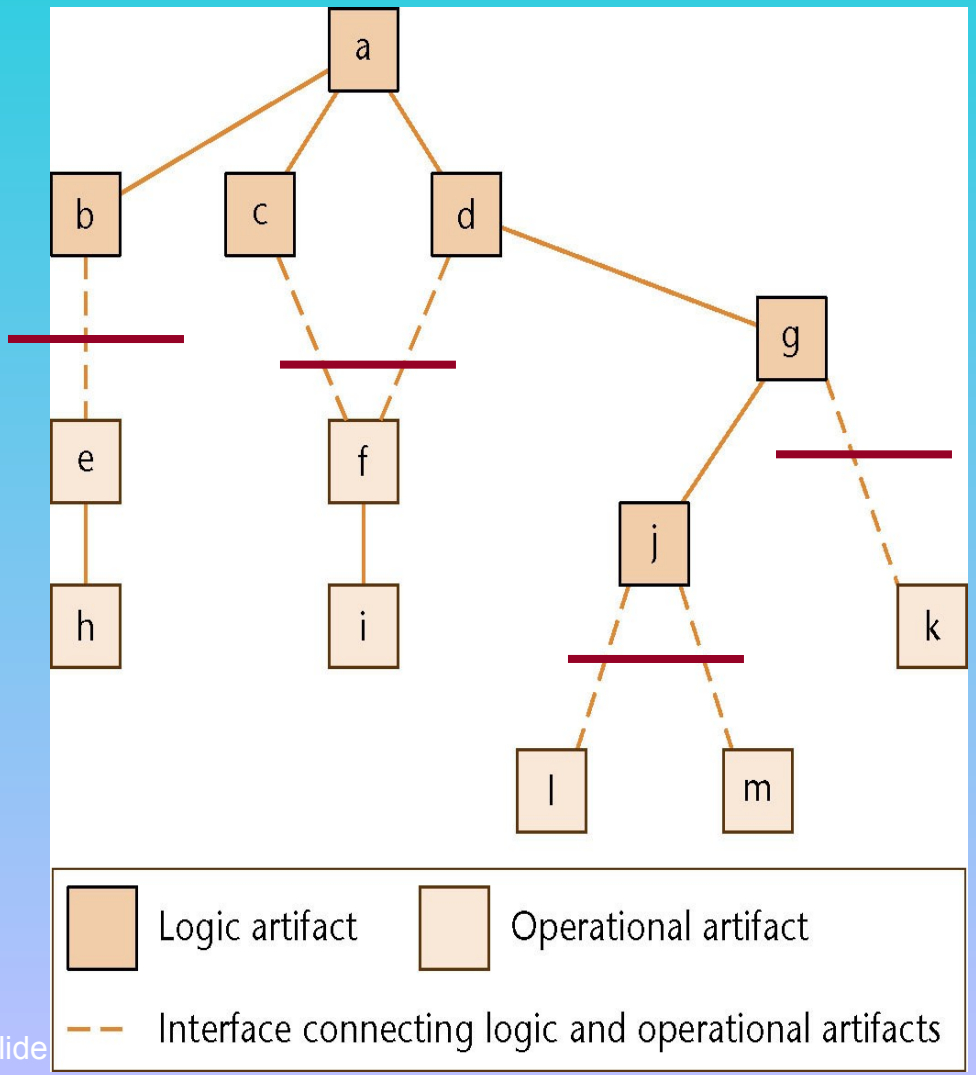


Logic artifacts include the decision-making flow of control

- In the example, artifacts a,b,c,d,g,j

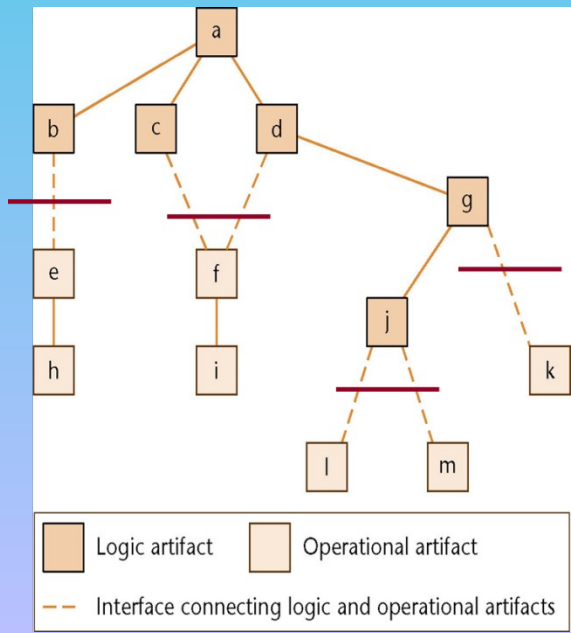
Sandwich Integration

- Logic artifacts are **Integrated Top-Down**
- Operational artifacts are **Integrated Bottom-Up**
- Finally, the **Interfaces** between the two groups are **tested**



Sandwich **Integration**

- Advantage 1
  - Major design **faults** are caught early
- Advantage 2
  - Operational artifacts are **thoroughly tested**
  - They may be **Reused** with confidence
- Advantage 3
  - There is **fault isolation** at all times



Summary

Approach	Strengths	Weaknesses
Top-down integration (Section 14.6.1)	Fault isolation Major design faults show up early	Potentially reusable code artifacts are not adequately tested
Bottom-up integration (Section 14.6.2)	Fault isolation Potentially reusable code artifacts are adequately tested	Major design faults show up late
Sandwich integration (Section 14.6.3)	Fault isolation Major design faults show up early Potentially reusable code artifacts are adequately tested	—

Figure 14.8

Integration of Object Oriented Products

- Object Oriented Implementation and Integration
  - Almost always **Sandwich** Implementation and Integration
  - Objects are **Integrated Bottom-Up**
  - Other artifacts are **Integrated Top-Down**

Sandwich integration (Section 14.6.3)	Fault isolation Major design faults show up early Potentially reusable code artifacts are adequately tested	—

EXAM 4 REVIEW

OO Implementation Workflow

From "blue print" to "house"/Code: One to One mapping

EstimateFundsForWeek::computeEstimatedFunds

public static void computeEstimatedFunds( )

This method computes the estimated funds available for the week.

float expectedWeeklyInvestmentReturn; (*expected weekly investment return*)

float expectedTotalWeeklyNetPayments = (**float**) 0.0; (*expected total mortgage payments less total weekly grants*)

float estimatedFunds = (**float**) 0.0; (*total estimated funds for week*)

Create an instance of an investment record.

Investment inv = **new** Investment ( );

Create an instance of a mortgage record.

Mortgage mort = **new** Mortgage ( );

Invoke method totalWeeklyReturnOnInvestment.

expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );

Invoke method expectedTotalWeeklyNetPayments (*see Figure 13.17*)

expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );

Now compute the estimated funds for the week.

```
estimatedFunds = (expectedWeeklyInvestmentReturn
      – (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
      + expectedTotalWeeklyNetPayments);
```

Store this value in the appropriate location.

MSGApplication.setEstimatedFundsForWeek (estimatedFunds);

// computeEstimatedFunds

OO I - JAVA

```
5   public static void compute()
6   //
7   // computes the estimated funds available for week
8   //
9   {
10    try
11    {
12      float expectedWeeklyInvestmentReturn = (float) 0.0; // expected weekly investment return
13      float expectedTotalWeeklyNetPayments = (float) 0.0; // expected total mortgage payments less total weekly
14      float estimatedFunds = (float) 0.0;
15
16      Investment  inv = new Investment ();    // investment record
17      Mortgage    mort = new Mortgage ();     // mortgage record
18
19      expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ();
20      expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ();
21
22      estimatedFunds = (expectedWeeklyInvestmentReturn - (MSGApplication.getAnnualOperatingExpenses ()
23                      / (float) 52.0) + expectedTotalWeeklyNetPayments);
24
25      MSGApplication.setEstimatedFundsForWeek(estimatedFunds);
26
27    }
28    catch (Exception e)
29    {
30        System.out.println ("***** Error: EstimatedFundsForWeek.compute () *****");
31        System.out.println ("\t" + e);
32    }
33  } // compute
```

OO Implementation Workflow

From "blue print" to "house"/ Code: One to One mapping

EstimateFundsForWeek::computeEstimatedFunds

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.

   float expectedWeeklyInvestmentReturn;          (expected weekly investment return)
   float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                  (expected total mortgage payments
                                                  less total weekly grants)
   float estimatedFunds = (float) 0.0;            (total estimated funds for week)
Create an instance of an investment record.
   Investment inv = new Investment ( );
Create an instance of a mortgage record.
   Mortgage mort = new Mortgage ( );
Invoke method totalWeeklyReturnOnInvestment.
   expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );
Invoke method expectedTotalWeeklyNetPayments          (see Figure 13.17)
   expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );
Now compute the estimated funds for the week.
   estimatedFunds = (expectedWeeklyInvestmentReturn
         – (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
         + expectedTotalWeeklyNetPayments);

Store this value in the appropriate location.
   MSGApplication.setEstimatedFundsForWeek (estimate
// computeEstimatedFunds
```

OO I – C++

```
void EstimateFundsForWeek::compute()
//
// computes the estimated funds available for week
//
{

   double expectedWeeklyInvestmentReturn = 0.0;      // expected weekly investment return
   double expectedTotalWeeklyNetPayments = 0.0;      // expected total mortgage payments l
   double estimatedFunds = 0.0;
   OperatingExpenses* op = new OperatingExpenses();

   Investment*  inv = new Investment ();      // investment record
   Mortgage*    mort = new Mortgage ();       // mortgage record

   expectedWeeklyInvestmentReturn = inv->totalWeeklyReturnOnInvestment ();
   expectedTotalWeeklyNetPayments = mort->totalWeeklyNetPayments ();

   estimatedFunds = (expectedWeeklyInvestmentReturn - (op->getAnnualOperatingExpenses ()
                    / 52.0) + expectedTotalWeeklyNetPayments);

   op->setEstimatedFundsForWeek(estimatedFunds);
```

"house" is 90% C

OO Implementation Languages

C++ vs. JAVA

OO I – C++

OO I - JAVA

does not matter the "blue print" is the same

having the "blue print" means 80% done coding

```
void EstimateFundsForWeek::compute()
//
// computes the estimated funds
//
{

 double expectedWeeklyInvestmentReturn = 0.0;        // expected weekly investment return
 double expectedTotalWeeklyNetPayments = 0.0;        // expected total mortgage payments
 double estimatedFunds = 0.0;
  OperatingExpenses* op = new OperatingExpenses();

 Investment*  inv = new Investment ();        // investment record
 Mortgage*    mort = new Mortgage ();         // mortgage record

 expectedWeeklyInvestmentReturn = inv->totalWeeklyReturnOnInvestment ();
 expectedTotalWeeklyNetPayments = mort->totalWeeklyNetPayments ();

 estimatedFunds = (expectedWeeklyInvestmentReturn - (op->getAnnualOperatingExpenses ()
                  / 52.0) + expectedTotalWeeklyNetPayments);

} // compute ()
```

"C++" is 90% C

or….

```
5   public static void compute()
12      float expectedWeeklyInvestmentReturn = (float) 0.0; // expected weekly investment return
13      float expectedTotalWeeklyNetPayments = (float) 0.0; // expected total mortgage payments less total weekly
14      float estimatedFunds = (float) 0.0;
15
16      Investment  inv = new Investment ();    // investment record
17      Mortgage    mort = new Mortgage ();     // mortgage record
18
19      expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ();
20      expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ();
21
22      estimatedFunds = (expectedWeeklyInvestmentReturn - (MSGApplication.getAnnualOperatingExpenses ()
23                       / (float) 52.0) + expectedTotalWeeklyNetPayments);
24
25      MSGApplication.setEstimatedFundsForWeek(estimatedFunds);
26
27  }
28  catch (Exception e)
31      System.out.println ("\t" + e);
32  }
33  } // compute
```

"JAVA" is 90% C

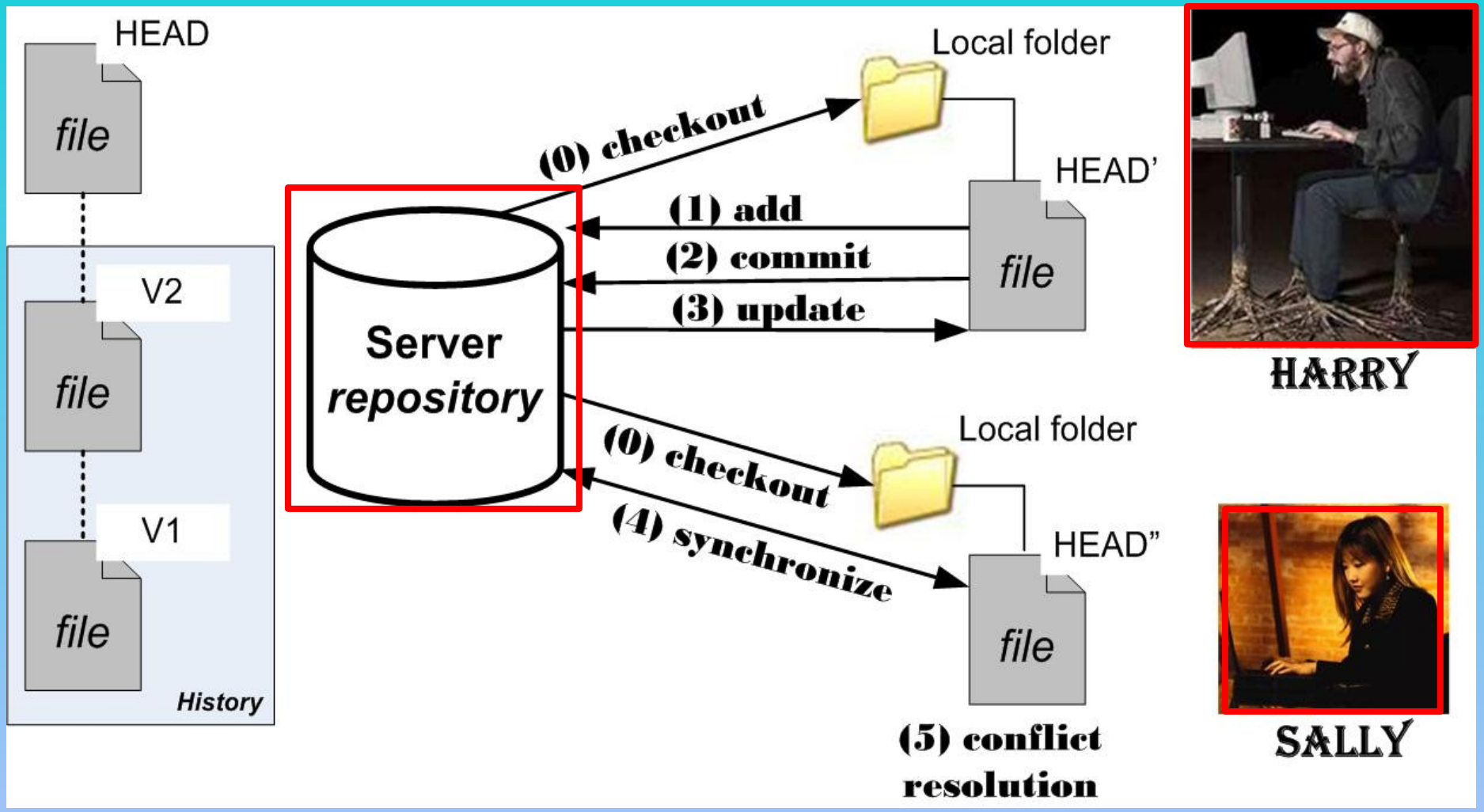
Management of Integration ("blue print" based)

- Example:
  - Design document used by **programmer P_1** (who coded code object $o1$) shows $o1$ sends a message to $o2$ passing 4 arguments
  - Design document used by **programmer P_2** (who coded code artifact $o2$) states clearly that only 3 arguments are passed to $o2$
- Solution:
  - The **Integration Process** must be run by the **SQA group**
  - They have the most to lose if something goes wrong

EXAM 4 REVIEW

SVN

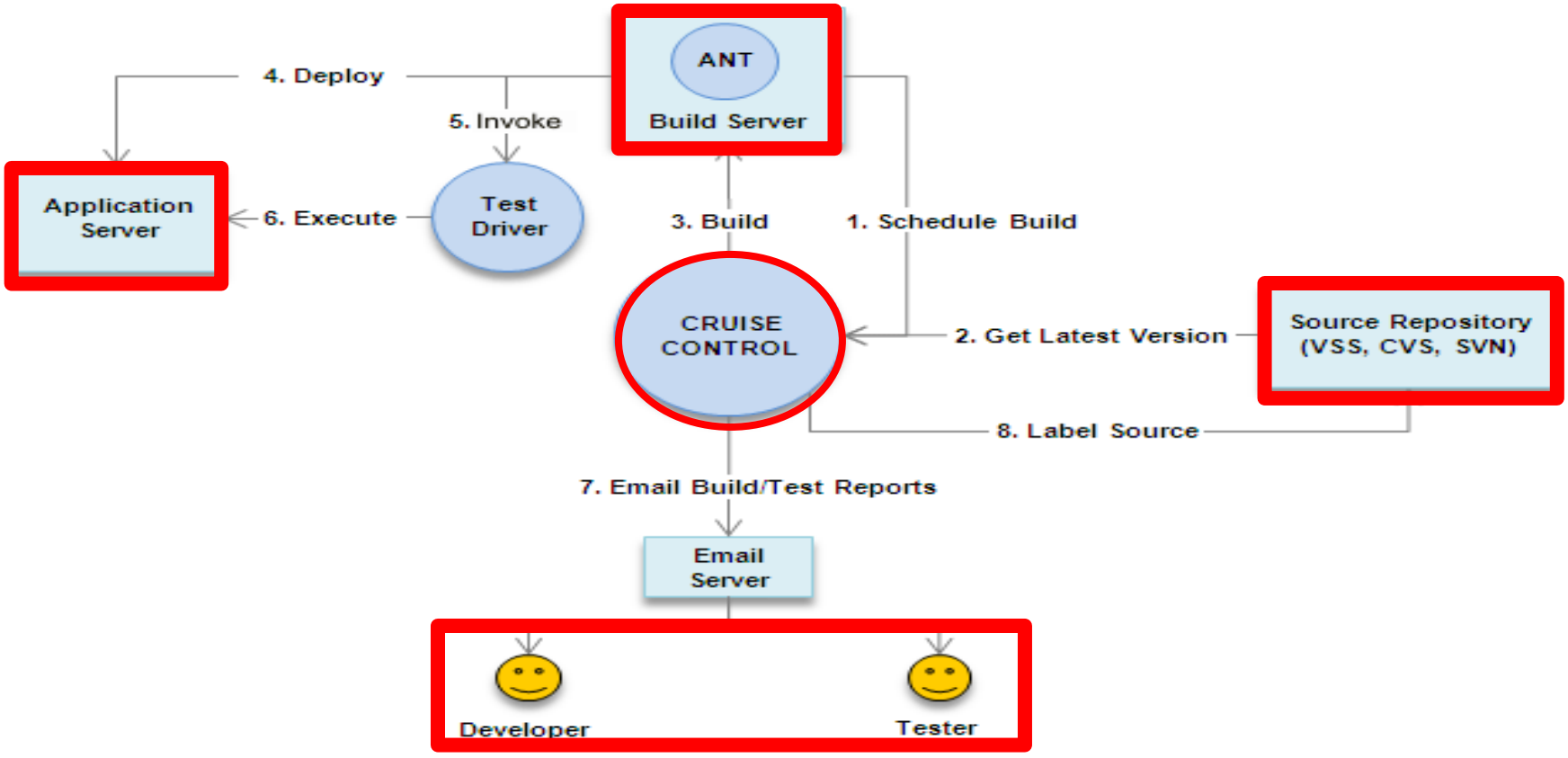
What is SVN - Subversion?

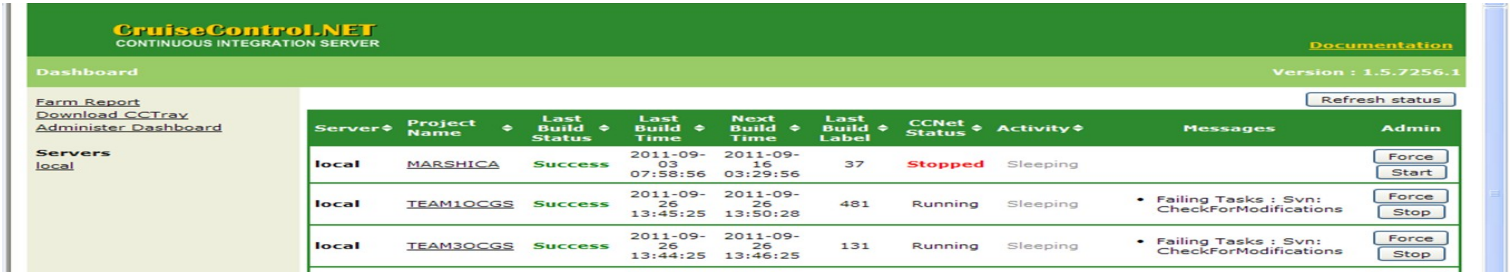


CASE Tools for Implementation

- **CASE Tools for Integration** include
  - Version-Control, Configuration-Control, and Build tools
  - Examples:
    - *rcs*, *sccs*, PCVS
    - **SVN, CCNET**

What is CruiseControl – Continuous Integration?





CASE Tools for the Complete Software Process

A **Large** Organization needs an **Environment**

A **Medium** Organization can probably manage with a **Workbench**

A **Small** Organization can usually manage with just **Tools**

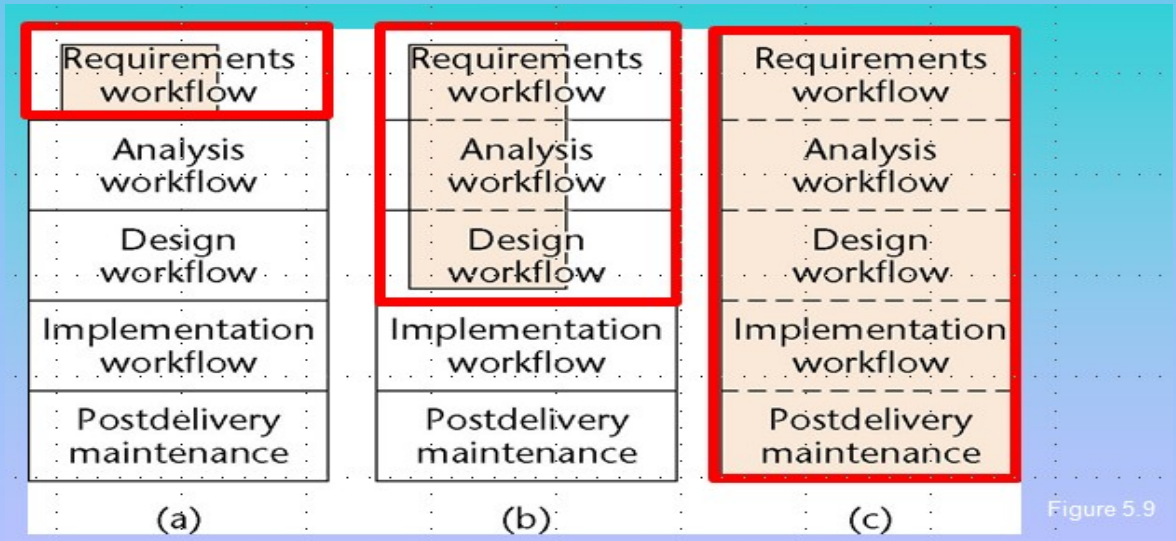


Figure 5.9

Tool versus (b) **Workbench** versus (c) **Environment**

Integrated Development Environments (IDE)

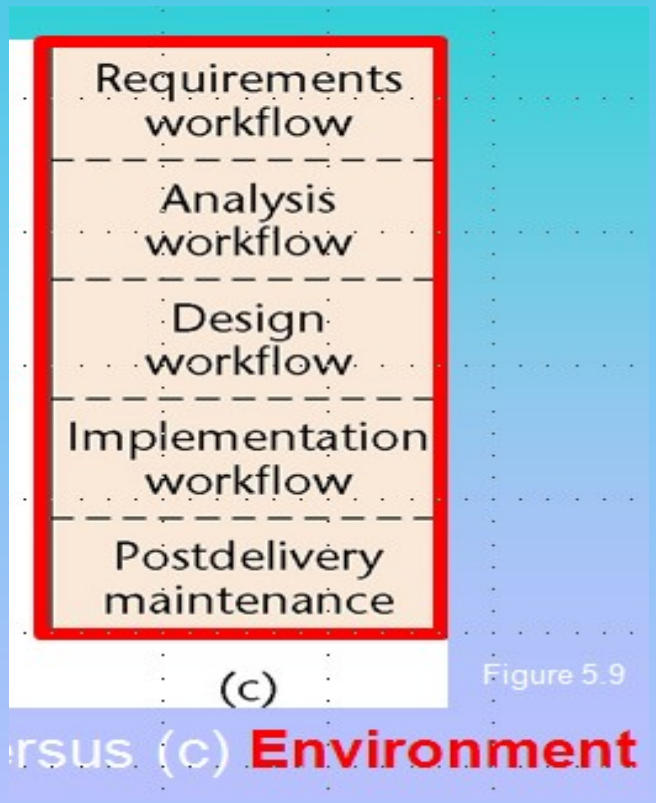
- The usual meaning of "**Integrated**"
  - User Interface **Integration**
  - Similar "look and feel"

  - **Visual Studio**, **NetBeans**, **Eclipse**, **RoR**, **Kohana**

- There are also other types of **Integration**
- **Tool Integration**
  - All **Tools** communicate using the same format
  - Example:
    - » **Unix Programmer's Workbench**

Process Integration

- The **Environment** supports one **specific Process**
  - **Target Process** - **Scrum**



Technique Based Environment

Subset: **Technique** Based Environment

- Formerly: “Method Based Environment”
- Supports a **specific Technique**, rather than a **complete Process**
- **Environment** exists for **Techniques** like
  - **Structured Systems Analysis**
  - **Petri Nets**

Technique Based Environment

- Usually comprises
  - Graphical support for Analysis, Design
  - A Data Dictionary
  - Some Consistency Checking
  - Some Management Support
  - Support and Formalization of **manual Processes**
  - Examples:
    - Analyst/Designer
    - Software through Pictures
    - **IBM Rational Rose**
    - Rhapsody (for Statecharts)
    - **MagicDraw**

Technique Based Environments

Advantage of a Technique Based Environment

- The **user** is forced to use one specific **method**, correctly

UML Modeling

Disadvantages of a Technique Based Environment

- The **user** is forced to use one specific **method**, so that the **method** must be part of the **Software Process** of that organization

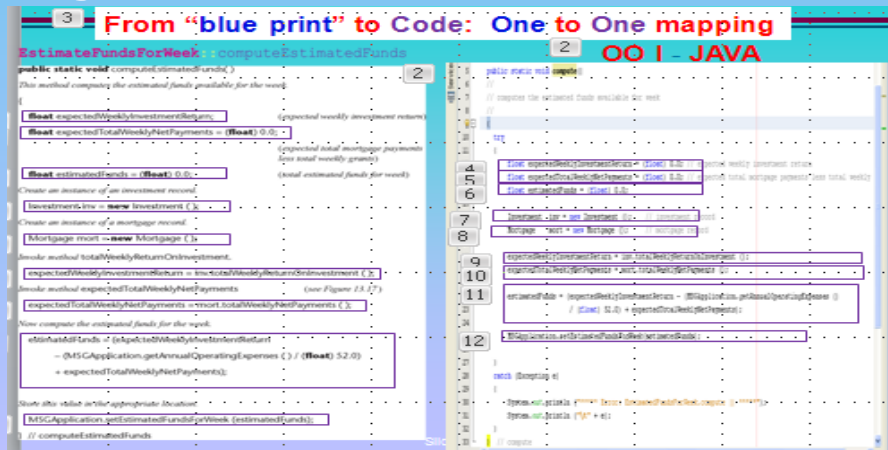
UML Modeling

Environments for Business Application

The emphasis is on **ease of use**, including

- A user-friendly GUI **generator**,
  - » Standard **screens** for **input** and **output**, and
  - » A **Code Generator**
    - **Detailed Design "blue print"** is the lowest level of abstraction
    - The **Detailed Design** is the input to the **Code Generator**

Use of this "programming language" should lead to a **rise in productivity**



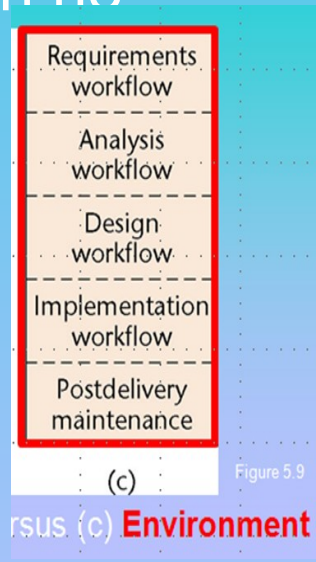
Potential Problems with **Environments**

No one **Environment** is ideal for all Organizations

- Each has its **strengths** and its **weaknesses**

Warning 1

- Choosing the wrong **Environment** **can be worse** than no **Environment**
- Enforcing a wrong **Technique** is **counterproductive**



Warning 2

- **Shun CASE Environments below CMM level 3**

(We **cannot automate a nonexistent Process**)

- However, a **CASE Tool** or a **CASE Workbench** is fine

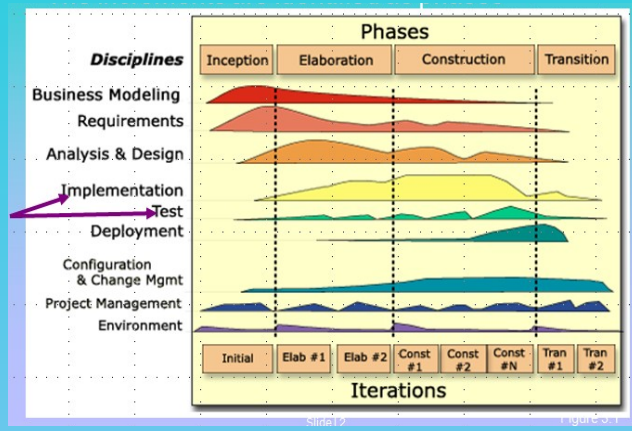
Metrics for the Implementation Workflow

The **five basic metrics** (size, cost, duration, effort, Quality)

plus

- **Complexity metrics**

Unified Process



Fault statistics are important

- Total number of **faults**, by types

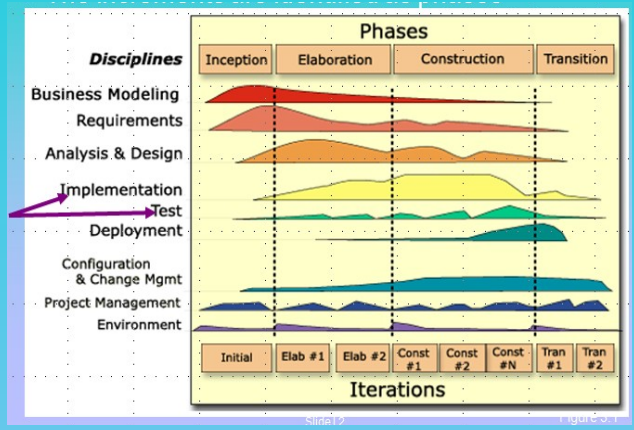
The **fault** data are incorporated into Checklists for **Code Inspections**

EXAM 4 REVIEW

Challenges of the Implementation Workflow

- **Management issues** are paramount here

Unified Process



- Appropriate **CASE Tools**

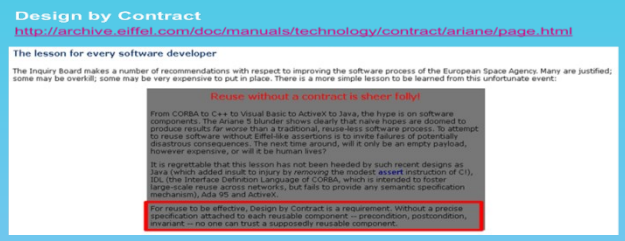
- **Communicating** changes to **all personnel**

SVN, CCNET

Challenges of the Implementation Workflow

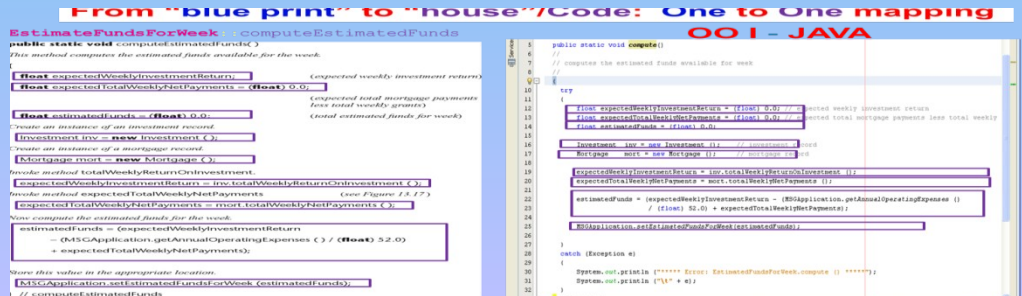
Code reuse needs to be built into **the product** from the very beginning

- **Reuse** must be a **Client Requirement (SRS)**
- The **S**oftware **P**roject **M**anagement **P**lan must incorporate **Reuse**



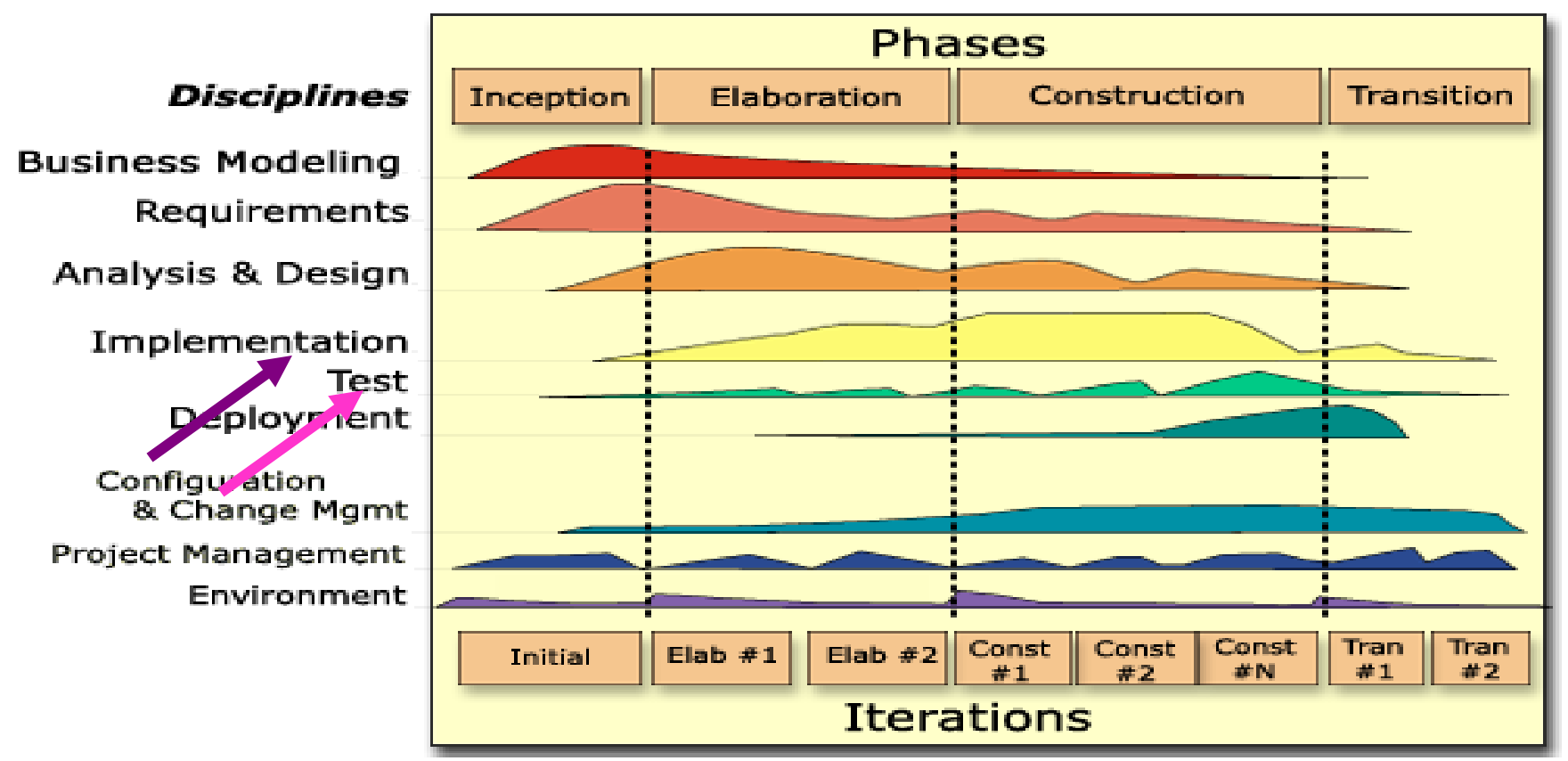
Implementation is technically straightforward

- **having the "blue print" means 80% done "house"/coding**



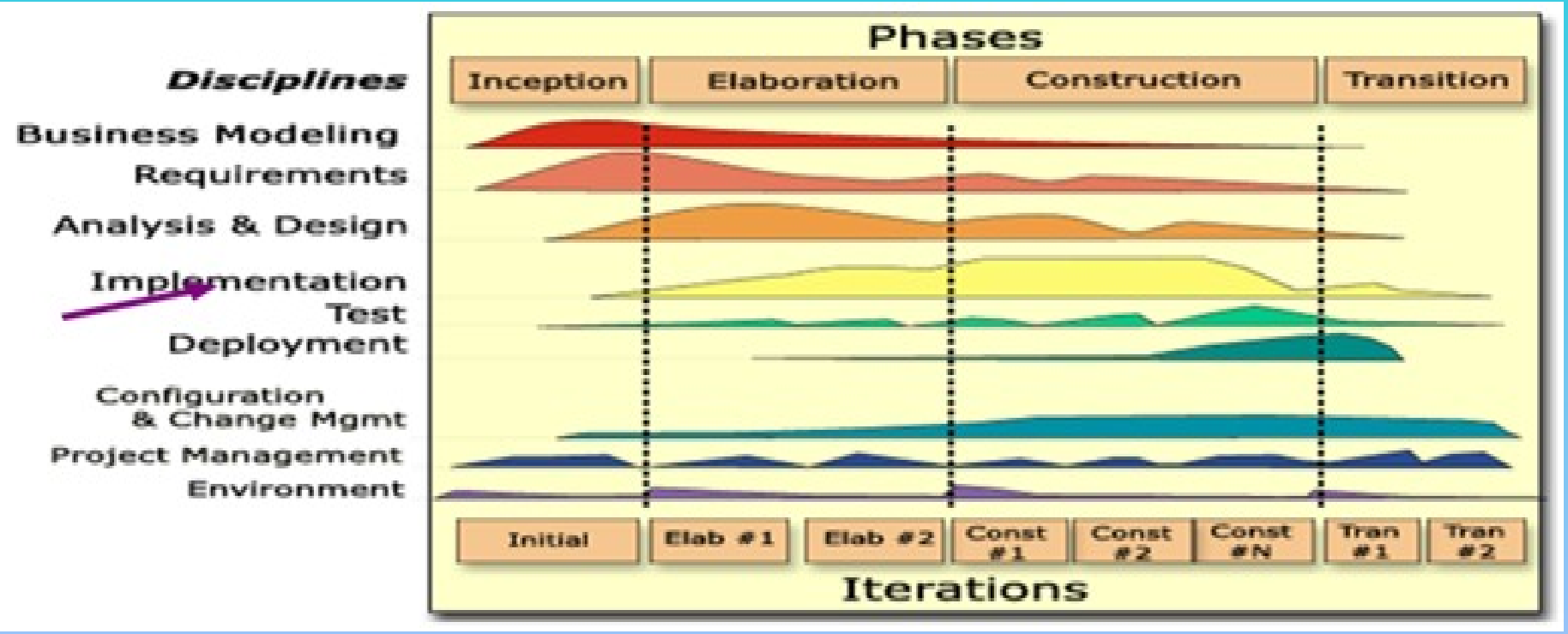
End **Implemetation** Workflow

- Next **Testing** **Workflow**



From 4:10 to 5:00 – 50 minutes.

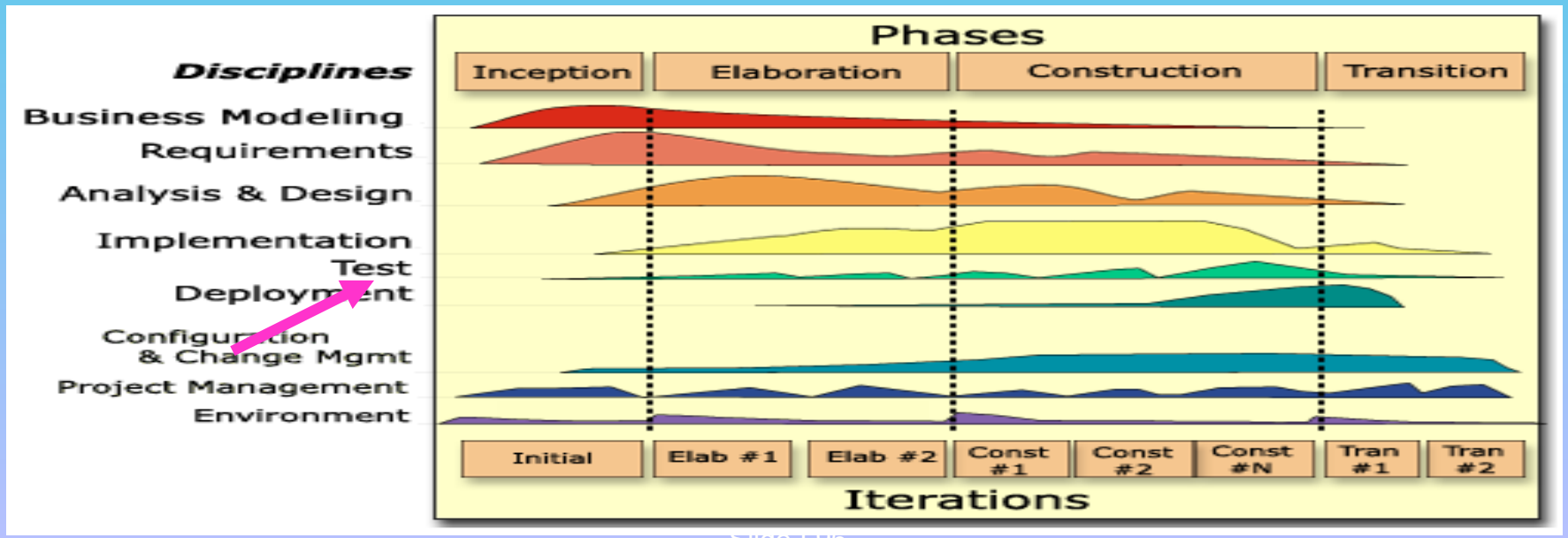
Implementation Workflow



THE END

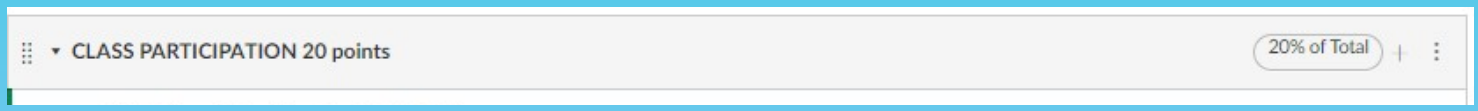
NEXT.

11.15.2023 (W 4 to 5:30) (25)		Lecture 9: Testing Tutorial 6 TDD			



From 5:05 to 5:15 – 10 minutes.

11.13.2023 (M 4 to 5:30) (24)		Lecture 8: **Implementation**			



PASSWORD: IN TEAMS

At 5:15 PM.

End Class 24

VH, Download Attendance Report
Rename it:
11.13.2023 Attendance Report FINAL

VH, upload Class 24 to CANVAS.