Module – II
Software Engineering Development Practices

Software Construction

Overview of Topics

Software Construction: Overview

'software construction' - detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging

- Software construction is strongly related to design and testing
  - Significant amount of design work is performed during construction itself
  - Both unit test and (some) integration testing is performed during construction
- Software construction is related to all other lifecycle processes
  - Requirements, maintenance etc.

[ref SWEBOK 4-1]

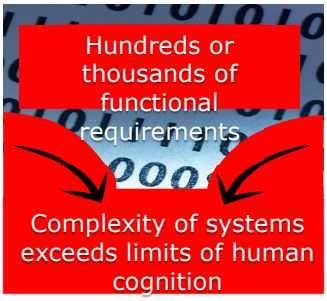
Software Construction: Overview

- Construction produces high volume of configuration items
  - E.g., source files, test cases etc.
- Construction is tool-intensive
  - relies heavily on tools such as compilers, debuggers, GUI builders etc.
- Construction is related to software quality
  - code is the ultimate deliverable of a software project
- Construction makes extensive use of computer science knowledge
  - in using algorithms, detailed coding practices etc.

[ref SWEBOK 4-1]

1. Software Construction Fundamentals

Minimizing Complexity



Creating simple, readable code
- Using standards
- Using specific coding techniques
- Using construction-focused quality techniques

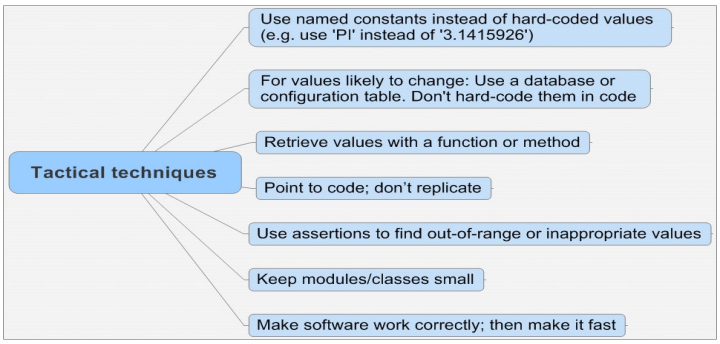
Reduce development process complexity
- Using patterns
- Reuse
- Keeping fan-in high and fan-out low
- Incremental development
- Error reduction techniques
- Heuristics

Minimize complexity in design
- Using 'enabling techniques' such as abstraction, decomposition & modularization etc.

ref 1-91

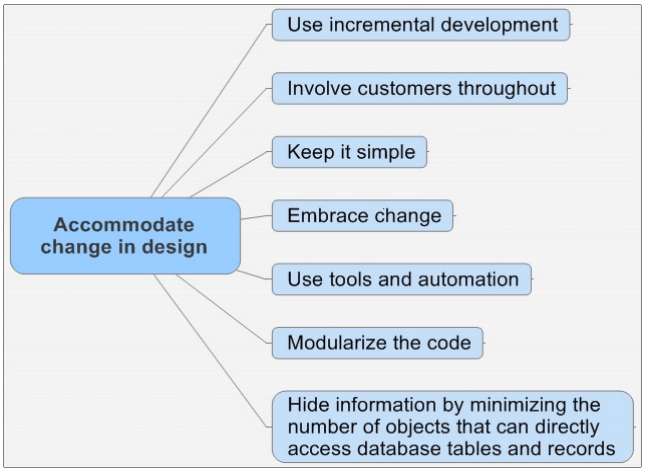
Anticipating Change

- Anticipating change affects how software is constructed
  - Hardware, business needs/process, laws & regulation etc. change.
  - We can use many tactical techniques to facilitate change:



 ref 1-92

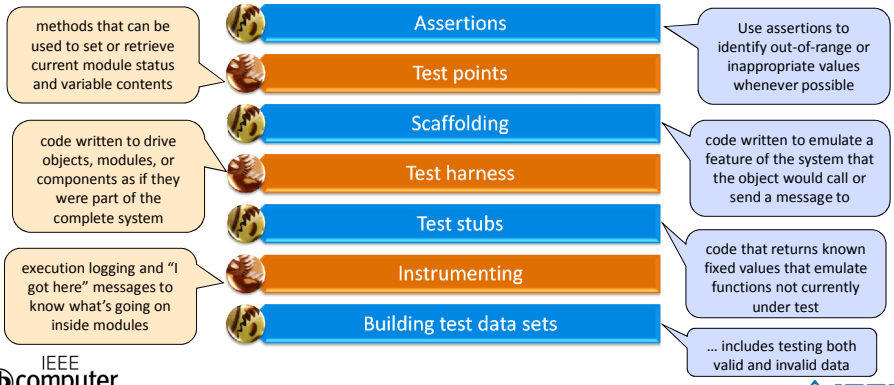
Anticipating Change …



 ref 1-93

Constructing for Testability

- We need to construct software to make it easy to test
  - … and find and fix bugs



ref 1-93

Standards in Construction

- Many standards originally identified during the requirements phase may need to be reviewed during development
  - Such standards can be business, regulatory, engineering, organizational, etc.
- Standards that may affect construction directly include:
  - Corporate development standards
  - Corporate policies related to working conditions and environment
  - Details in any published standard that will be implemented in the code
    - E.g., *IEEE Std. 1284-1994* (standard signaling method for a bidirectional parallel peripheral interface for personal computers)
  - Standards dealing with interfaces with systems whose operation is specified by external (commercial or industry) standards
    - These may be discovered in requirements or design and should also be reviewed prior to or during construction to ensure that the code adheres to these constraints

ref 1-94

Development Standards

- Coding standards
  - For uniform formatting, naming conventions, etc. in an organization
- Standards for communication methods
  - such as project reporting standard
- Programming language standards
  - For example, C++ language standard
- Platform (hardware, software, OS, etc.) standards
  - such as programmer interface standards for OS calls – example, POSIX
- Tool standards
  - Restrict specific tools to promote familiarity and consistency for the developers; for example, allowing only one UML modeling tool enables sharing of designs between projects and reuse over time
- Web standards
  - Such as W3C consortium (World Wide Web Consortium)

ref 1-95

Discussion Question

To test the software, you added execution logging and "I got here" messages to know what's going on inside modules. This technique is called:

A. Assertions
B. Test points
C. Test stubs
D. Instrumenting

Answer D: Instrumenting

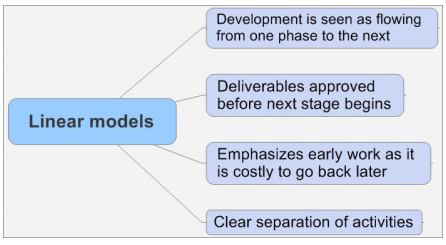
ref 1-94

2. Managing Construction

Construction Models: Linear Models

- Linear models treat construction as an activity which occurs only after significant prerequisite work has been completed
  - including detailed requirements work, extensive design work, and detailed planning
- Linear models tend to emphasize the activities that precede construction (requirements and design)
  - and tend to create more distinct separations between the activities

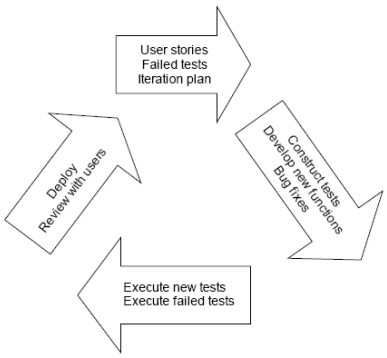
Examples: waterfall and staged delivery lifecycle models



  ref 1-96

Construction Models: Iterative Models

- Design & test activities intermingled with construction
  - Short phases
- Effective for business application development
  - Where requirements may change rapidly
  - Working (incomplete) software delivered quickly to customers
  - Feedback from customers is fed into the next cycle
- Examples
  - Rapid Application Development (RAD)
  - eXtreme Programming (XP)

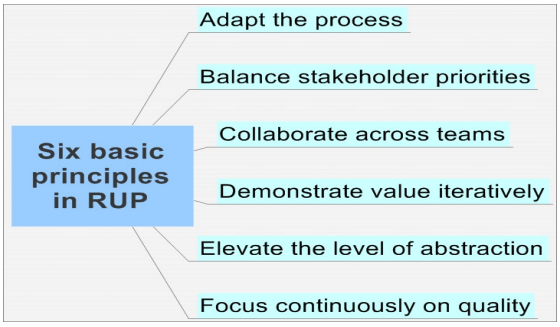


XP Iterative Development

ref 1-96

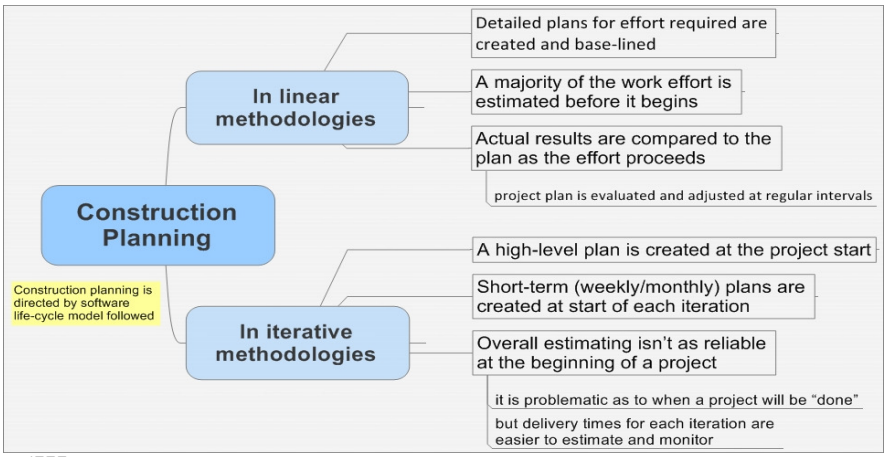
Rational Unified Process (RUP)

- RUP is a process framework
  - tailored by the development team
  - phases overlap, with requirements engineering continuing after design has started



ref 1-96

Construction Planning



  ref 1-98

Construction Planning: Linear SLCs

- Linear software life-cycle (SLC) models are best-suited for projects
  - with stable requirements, straightforward design, low risk etc.
  - under maintenance, or
  - that have fixed delivery dates (like tax season, a national coordinated launch date, or a regulatory or contractual deadline)
- Linear SLCs benefit from these strategies:
  - Complexity managed via the design
  - Stable requirements that help ensure minimal changes during development
  - Frequent builds, with verification done at each level (unit, integration, system) and integration testing after each build

  ref 1-98

Construction Planning: Iterative SLCs

- Iterative SLCs are well-suited for projects …
  - where requirements are not well understood, the design is complex or challenging, risk is high or the development team is unfamiliar with the applications area
- Iterative SLCs benefit from these stratagies:
  - Complexity managed by developing small pieces at a time for use and review and using simple designs that isolate changes to the system
  - Change built into the process (customers use working software and request changes for future iterations)
  - Verification testing that is less formal and may include test-first (writing the test before the code), incremental test development, or user testing

ref 1-98

Discussion Question

Which of the following MOST LIKELY describes a project that was run using an iterative software life cycle model?

A. Activities for requirements, design, construction etc. were clearly identified and separated

B. Overall estimate at start of project was way off from actual development time

C. Project had very stable requirements

D. Construction preceded detailed requirements work, extensive design work, and detailed planning

Answer B: Overall estimate at start of project was way off from actual development time

Construction Measurement

Measurement system focuses on two areas:

Proceeding as planned?

Typical measures: effort expenditure and rate of completion

Adjustments to plan based on milestones beaten, met or missed

Technical quality?

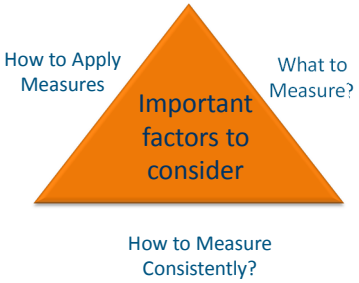
Typical measures: Lines of code developed, number of defects found in testing etc.

Can be useful when adjusting the construction plans and processes

[ref 1-99]

Common Technical Measures

- Number of components built and the rate (components per week or month)
- Number of requirements implemented in the component
- Number of requirements changed
- Number of test cases developed
- Lines of code developed, modified, reused, or destroyed
- Code complexity (where measures range from lines of code to cyclomatic complexity)
- Code inspection and review statistics
- Number of defects revealed by testing



[ref 1-99]

Construction Measurement: Cautions

- All measurements should be used with caution
  - Counting classes is an uncertain measurement because different classes will show differences in areas such as:
    - Sizes (LOC or SLOC), numbers of methods or attributes, number of inherited methods, number of fan-ins, functional complexity etc.
  - A low defect rate may be due as much to insufficient testing as it is to good coding
    - (Defects per week is a simple rate. Defects divided by LOC, SLOC, or class count yields a defect density value)
  - Measuring developer productivity by lines of code per week can lead to problems
    - Would encourage developers to unnecessarily long code segments or duplicate code segments

 ref 1-99

Discussion Question

Software metrics should be evaluated for their utility in certain areas of application. Which one of the following areas of application should NOT be considered when evaluating the utility of software metrics?

A. Determining product complexity
B. Determining productivity of individual staff members
C. Determining when a desired state of quality has been achieved
D. Determining the validity of project processes

… Discussion Question

Answer B: Determining productivity of individual staff members

Rationale: Use of metrics for personnel evaluation distorts the measurements made because people will try to make the numbers look good. This prevents the use of the measurements for all of the other mentioned purposes. Other writers also make the point that software metrics should not be used to determine personnel actions.

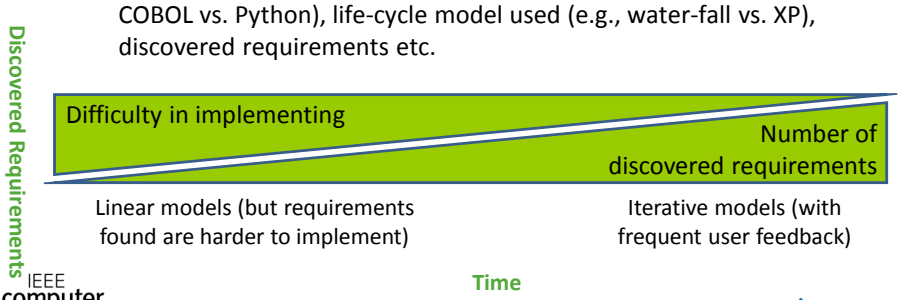
A, C, and D appear on a list on [SE-BestPractices]. Other items on that list include: Tracking project and product progress, Analyzing defects, Determining project and product complexity, and Providing a basis for estimating future projects.

[SE-BestPractices] Christensen, Mark, Thayer, Richard, *Project Manager's Guide to Software Engineering's Best Practices*, Wiley - IEEE Computer Society Press, 2002, p449

3. Practical Considerations

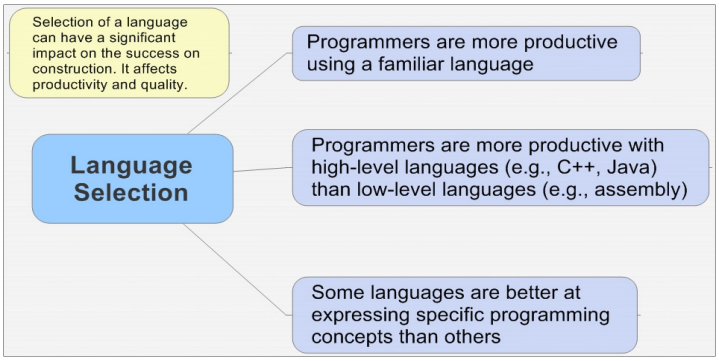
Construction Design

- Some detailed design work will occur during construction
  - The amount will vary by project to a level of detail that allows coding to begin.
  - Getting to the right level of detail may take many iterations of decomposition, depending on programming language used (e.g., COBOL vs. Python), life-cycle model used (e.g., water-fall vs. XP), discovered requirements etc.



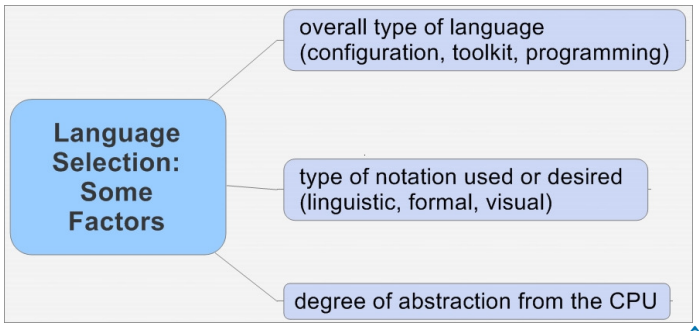
Construction Languages

- Construction languages: mechanisms adopted to translate the intent of the designers into machine-readable form



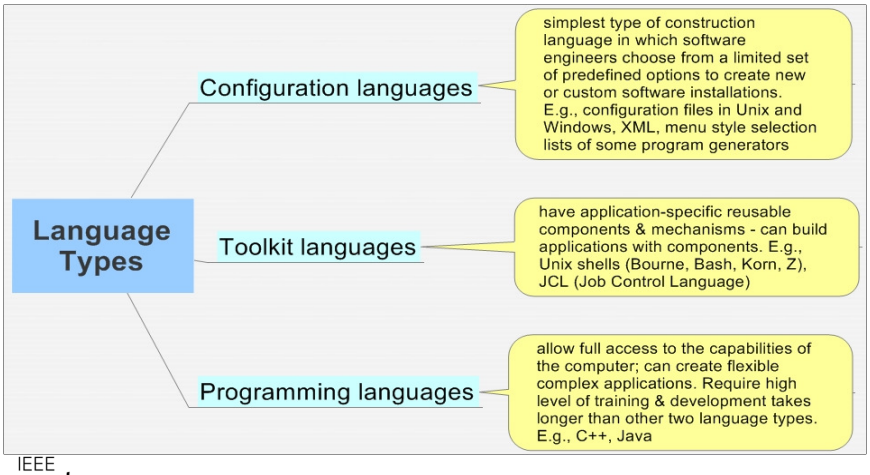
Language Selection: Factors

- There are many factors to consider in selecting a construction language
  - See figure for three such factors



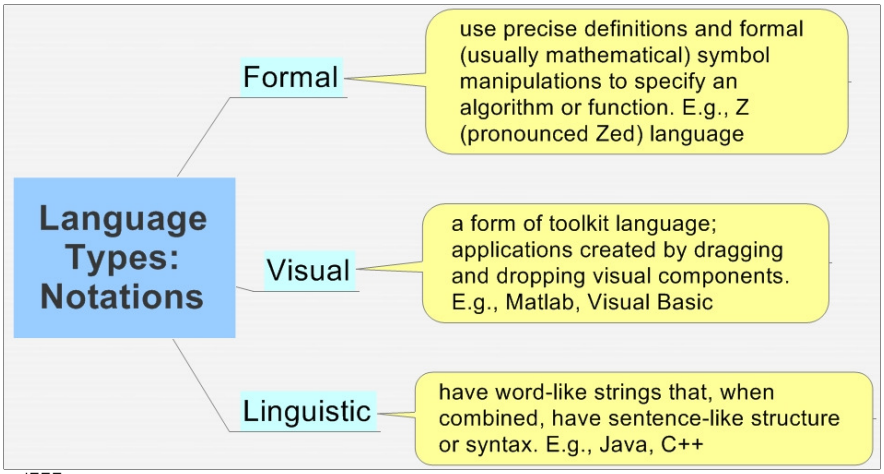
ref 1-101

Construction Languages…



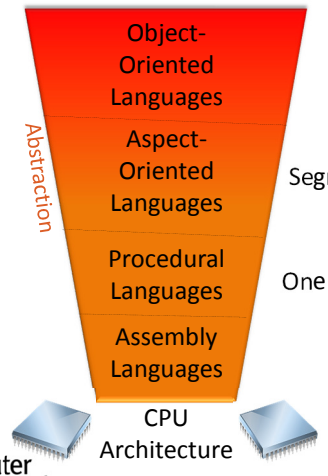
ref 1-102

Construction Languages…



 [ref 1-102]

Degree of Abstraction



Consider problem in terms of objects

Segregate aspects (concerns) from core functions

One statement conveys multiple CPU operations

One statement per CPU operation

 [ref 1-103]

OO Languages – Examples

Smalltalk

- Pure OO language
- Reusable class library (fast)
- Iterative, incremental development
- Not compiled; virtual machine (portable)

C++

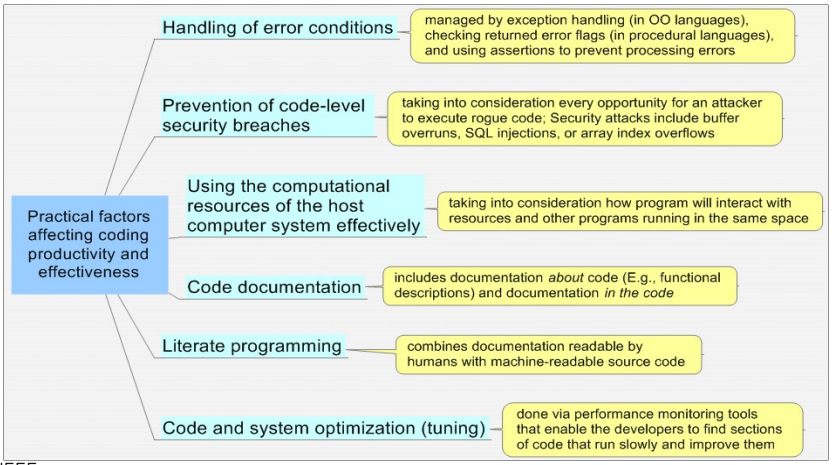
- Multi-paradigm
  - Supports imperative, OO and generic paradigms
- Compiles to native executable code (for each platform)
- Sometimes, results can be different in different platforms (less portable)

Java

- Compiled & Interpreted
  - Code compiles to byte code, runs on virtual machine (VM)
  - VM for each CPU (device independent)
- Write code only once rather than for each platform (portable)

ref 1-104

Coding



ref 1-106

Literate Programming

- Literate programming
  - combines documentation readable by humans with machine-readable source code in the same file
  - purpose is to keep the comments and the code closely connected
  - this approach helps in understanding the code better
    - with literate programming, someone who has not previously worked with the code can quickly comprehend what the function is doing, what cautions have already been thought of, and why some action is being done the way it is
- An example of literate programming is JavaDoc tool
  - It is a Java utility from Sun that will extract comments and create documentation for the developer
    - Provided that the developer has provided documentation comments in code

[ref 1-106]

Reuse

- Reuse vs. salvaging
  - Reuse – using software that is designed for reuse
  - Salvaging – using software that is *not* designed for reuse

"Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle"
- [IEEE1517-99]

IEEE Std 1517-1999, IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes: IEEE, 1999

[ref 1-108]

Reuse Levels

- Reuse levels
  - Reuse may apply at the system, component, and object levels
- System level reuse
  - When an application is incorporated into another system without change
- Component level reuse
  - Internally developed components likely to be reused if they are application-specific
    - However, designing for reuse or modifying a component to make it more reusable may be feasible
- Object level reuse quite common in the form of standard libraries (e.g., mathematical functions)

  [ref 1-108]

Benefits of Reuse

- Reduced development costs
  - fewer components need to be specified, designed, implemented, and validated
- Quicker time to market
  - both development and validation time can be reduced
- Increased dependability
  - the software has been tried and tested
- Reduced process risk since the cost is known
- Allows specialists to develop software
  - which encapsulates their knowledge
- Standards compliance
  - some standards can be implemented as a set of standard reusable components

  [ref 1-108]

Disadvantages of Reuse

- Increased maintenance costs
  - if the source code is not available
- Lack of tool support
  - if the software process does not take reuse into account
- Difficulty associated with organization wide reuse
  - in cataloging, archiving, and retrieving reusable assets across business units
- Limited documentation of the components
- Perception of application developers
  - developers may perceive "top-down" reuse efforts as an indication that management lacks confidence in their technical abilities
- Dependency on language features for fostering reuse
  - such as inheritance, polymorphism, templates, exception handling etc.

  ref 1-108

Planning for Reuse

- Reuse must be carefully planned
  - The development schedule is a key factor. If the software must be delivered quickly, use of off-the-shelf components may be appropriate
- Some factors leading to successful reuse:
  - Supporting iterative development processes and building the reusable components incrementally
  - Ensuring high-level management support for the reuse program
  - Taking advantage of personnel continuity between old and new programs
  - Making reuse an integral part of the development process
  - Estimating cost and savings realistically

  ref 1-109

Construction Quality

- Activities for ensuring construction quality include:

The process of writing code (in a framework or test harness) that will call the methods or functions in the code module under test and check the results

Peer review

formal name for a common and informal process where the developer shows the code to other developers, looking for advice or comment

Unit testing

Code being tested or reviewed is executed one statement at a time, and the debugger shows the values of variables and the flow of control as each statement executes

Test-first

The practice of designing and building the test harness before writing the code that will be tested. Balances developers tendency to write tests to match their code

Code stepping

ref 1-110

Construction Quality…

- Activities for ensuring construction quality include:

The process of analyzing code to locate a known defect

Pair-programming

Two developers work on same code, with one focusing on function logic and another on syntax and accuracy

Debugging

Any process or tool that examines the code without executing it. E.g., peer review and code Inspections, static analysis tools

Code Inspections

A formal process of peer review. Developer presents code for review to code inspectors

Static analysis

ref 1-111

Integration

- Integration: combining components into a subsystem or system
- Keys to successful integration:
  - good design, configuration and source code management, a defined integration process etc.
- Some integration issues:
  - How items should be prepared and qualified for integration, the degree of testing, and quality measures criteria
  - Hardware, software dependencies between systems/subsystems
  - When items should be combined and in what order
- Integration in software development models
  - Linear models tend to have long intervals between integration phases
  - Incremental models rely on frequent integration phases

  [ref 1-109]

Construction Testing

- Objective of construction testing
  - reduce the gap between the time at which faults are inserted into the code and the time those faults are detected
- Construction testing
  - involves mainly unit testing
    - Some integration testing is also done (it is technically testing phase activity)
    - Regression testing is done only at the unit level during construction
  - done by software engineer who wrote the code
  - does not typically include system testing, alpha testing, beta testing etc.
- When is construction testing performed?
  - generally performed after code has been written
  - in many cases, test cases may be created before code is written (e.g., in Test Driven Development)

  [ref SWEBOK 4-4]

Discussion Question

What is "test first" approach - why would one consider writing tests before writing the code?

  ref 1-110

Discussion Question

Which of the following programming language notations is based on precise, unambiguous, and mathematical definitions?

A. Linguistic notations
B. Formal notations
C. Visual notations
D. None of the above

  ref SWEBOK 4-3

… Discussion Question

Answer B: Formal notations

Rationale:

"Formal notations rely less on intuitive, everyday meanings of words and text strings, and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions.

Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language.

Formal constructions also use precisely defined ways of combining symbols which avoid the ambiguity of many natural language constructions." [SWEBOK]

 ref SWEBOK 4-3

Discussion Question

A new software engineer is added to a project. Which of the following practices followed in code would BEST help the new engineer understand the code better?

A. Defensive programming techniques designed to use the host computer system's resources effectively
B. Company formatting and naming standards for use with the required procedural language's exception handling
C. Refactoring the code
D. Literate programming such as JavaDoc

Answer D: Literate programming such as JavaDoc.

Discussion Question

The purpose of _____ is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected

A. Usability testing
B. Installation testing
C. Construction testing
D. Regression testing

Answer C: Construction testing

 [ref SWEBOK 4-4]

4. Construction Tools

Development Environments – I

- Limited choice in selecting development and implementation environments
  - restricted by the choice of language, methodology, application, and development and target hardware platforms
- Some factors in selecting development environments:
  - Commercial vs. open-source
    - Developers often prefer open-source tools, but some organizations think that open-source tools are untrustworthy or less capable than commercial tools
  - Support of development process
    - Support of the development process includes ensuring that tools such as debuggers, test builders, and analysis tools are available
      - An issue in obscure hardware and operating systems

ref 1-113

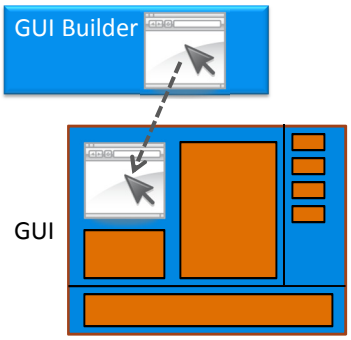
Development Environments – II

- Some factors in selecting development environments:
  - Security
    - Framework, language, and tools may affect vulnerability to external attack
  - Access to future capabilities
    - Future capabilities of tools should also be taken into account when considering which tools to use
  - Size of product and distribution of development team
    - Large projects are more complex and require a larger development environment
  - Degree of integration among /between tools and environment
    - Integration among tools and between tools and the environment enhances productivity by enabling or easing activities such as tracing requirements to code segments, generating skeleton code from models, or debugging

ref 1-113

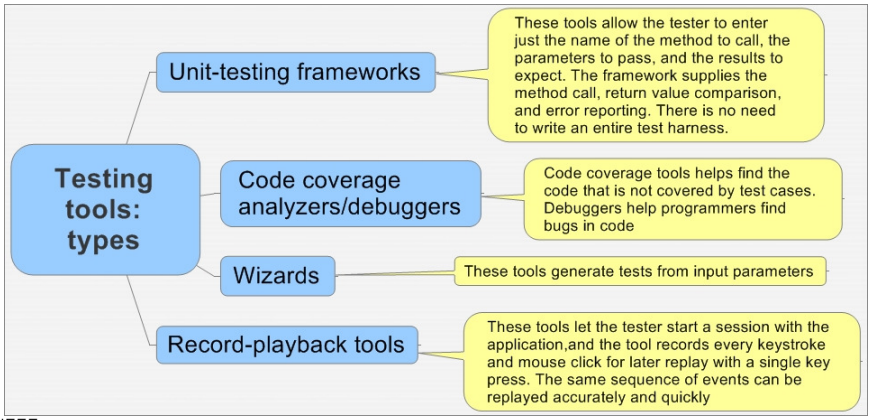
GUI Builders

- GUI builders: software tools that simplify building the GUI
- Advantages
  - Construction speed: Faster to drag-and-drop components than writing code
  - Ease of construction: Allows developer to focus on UI aspects than worrying about low-level code
- Disadvantages
  - Static code: Difficult to integrate programmer code into generated code
  - Often generates inefficient code
  - Generated code is difficult to modify



ref 1-114

Unit Testing Tools



ref 1-115

Automated Unit Testing: Advantages

- Unit-testing tools remove some of the manual effort required to write the test framework code
- Some advantages of automated unit testing:
  - There is a lower chance of executing the test incorrectly
  - Once automated, the test is available for the rest of the project
  - Automated tests can be run frequently. They should be run after every build
  - Tools improve the chances of finding problems at the earliest possible moment

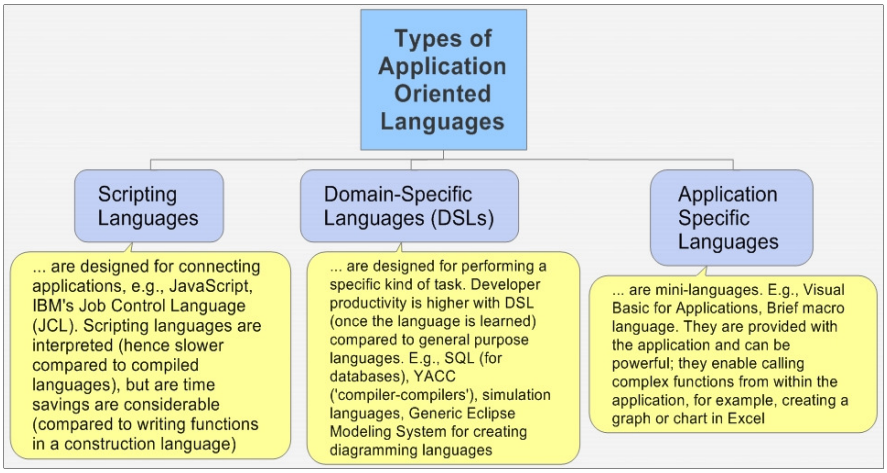
  ref 1-115

Application-Oriented Languages

- Specialized for specific purposes/problem domains
  - Differ from the construction languages (like C++, Java etc)
    - AOL provides direction to an existing complex application
    - Construction languages provide direction to an OS
- Simplify development
  - By abstracting complex but repeatable activities into statements understandable to a compiler
- Can be pre-existing or developer created
  - Pre-existing language examples: ALHARD (h/w simulation), Perl
  - Developer created languages: created using “compiler compilers” like YACC and BISON

  ref 1-117

Application Oriented Languages: Types



  ref 1-115

Discussion Question

Which of the following are NOT Application Oriented Languages?

A. Scripting languages (such as JavaScript and JCL)
B. Construction languages (such as C++ and Java)
C. Domain-specific languages (such as SQL, YACC, BISON)
D. Application specific languages (such as Visual Basic for Applications)

Answer B: Construction languages (such as C++ and Java)

Construction languages are different from Application Oriented Languages. AOLs provide direction to an existing complex application; construction languages provide direction to an OS.

5. Construction Technologies

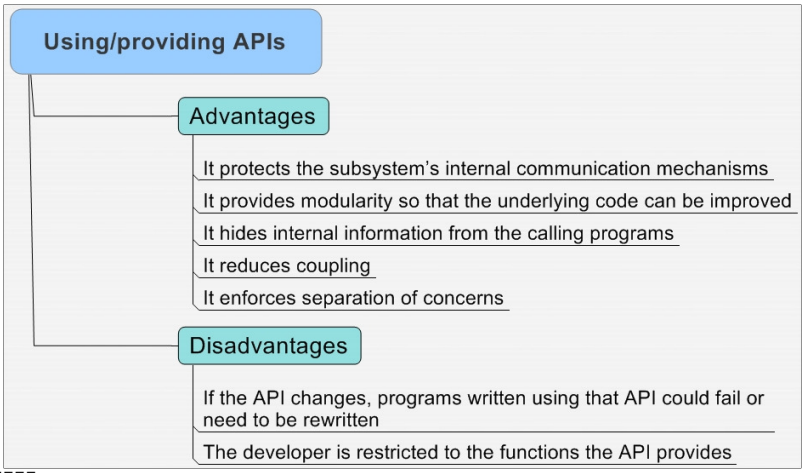
API Design and Use

API: A code interface that a OS/library exposes to handle requests for services from computer programs using it

- API (Application Programming Interface) is an abstraction
  - It specifies behavior of the functions and parameters specified
    - but does *not* specify how the behavior is implemented
- Open/Closed APIs
  - Open APIs – detailed information is available publicly
    - E.g., Microsoft APIs for Windows
  - Closed APIs – detailed information not available publicly
    - When APIs are for sale to generate license revenue

ref 1-116

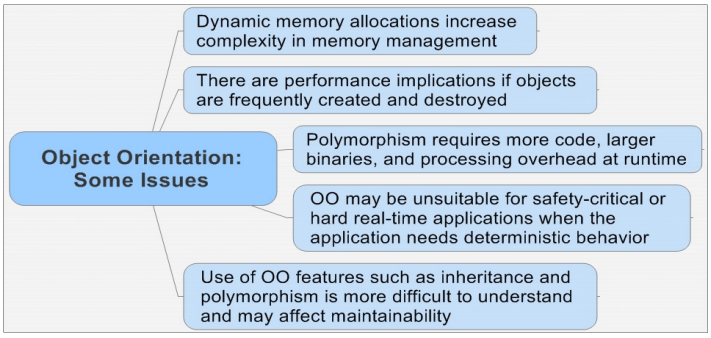
APIs: Advantages and Disadvantages



 ref 1-117

Object Orientation: Runtime Issues

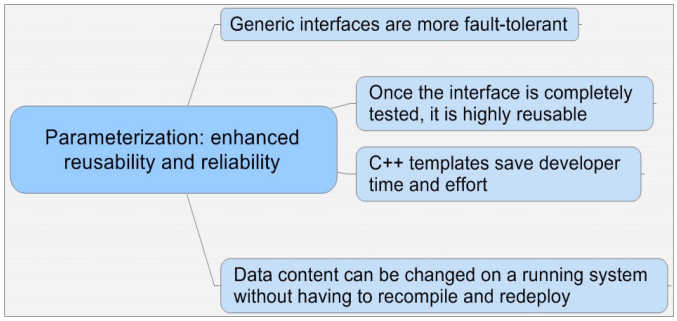
- In OOP, the advantages of abstraction, inheritance, and message passing usually outweigh the disadvantages
  - However, it is important to understand the disadvantages also (see figure for some issues)



 ref 1-117

Parameterized Types (Generics)

- Some OO languages (e.g., C++, Java) support parameterized types/generics
  - enables writing classes that can be called without consideration of the data type passed to the interface



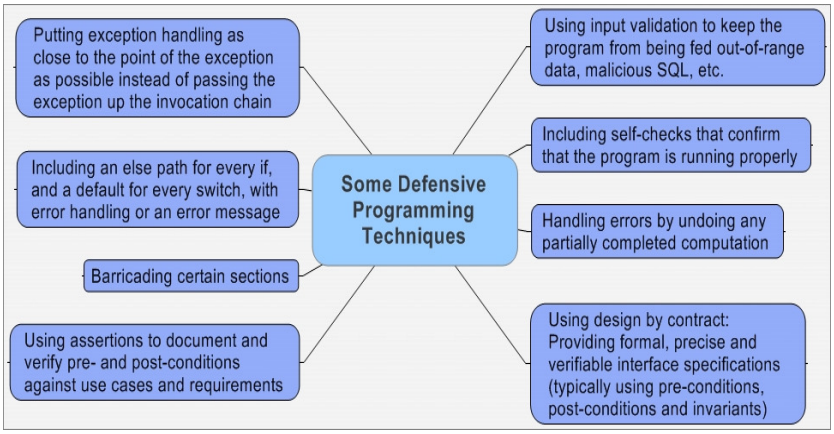
 ref 1-118

Defensive Programming - I

- What is defensive programming?
  - A set of development techniques that anticipate certain kinds of problems during program execution and attempt to prevent or handle them
- What is the intention for following this technique?
  - To improve the chances that the software will continue to function inspite of unforeseen use
- What is the goal?
  - To have the application issue an error message (‘graceful exit’) instead of executing incorrectly or crashing

 ref 1-118

Defensive Programming - II



ref 1-118

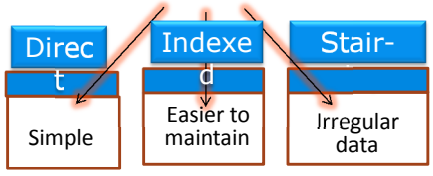
Table-driven Techniques

- Table-driven methods
  - Implement a lookup of information in a table rather than using logic statements
  - Particularly useful when the logic chain becomes long and complex
  - Used appropriately, table-driven methods make the code simpler, easier to modify, and more efficient.
  - Two issues to consider are: how to index the entries and what to store

Table-Driven Methods

Long and complex logic chain replaced by table lookup

If lookup result is data, the data can be stored.
If lookup result is an action, code or reference to a routine can be stored

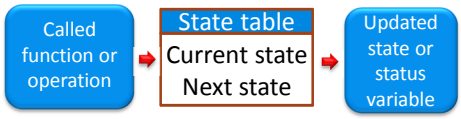


ref 1-119

State-based Techniques

- State-based methods
  - use a state or status variable (a number) that is changed by each called function or operation to indicate the new state
  - status variables - which reflect the state of a program - tend to change more frequently than other data
- State tables are similar to truth tables
  - inputs are the current state
  - outputs include the next state, along with other outputs

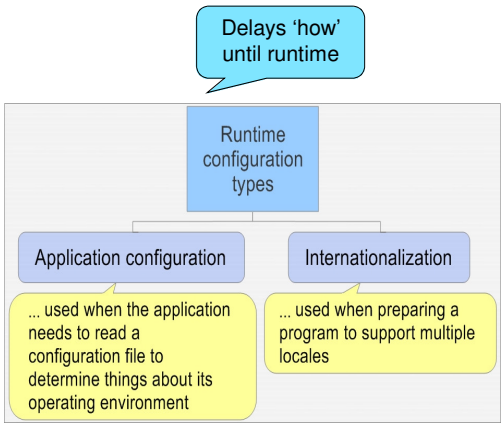
State-Based Methods



ref 1-119

Runtime Configuration

- Runtime configuration is a development technique
  - delays decisions about "how" to do something until the program is executed
  - enables a program to adjust at runtime
    - for different platforms, variations in hardware, different languages, different character sets etc.



ref 1-119

Application Configuration

- "Configuration file"
  - Used to specify parameters for runtime configuration of the application
- E.g. Linux/Unix use large number of such configuration files
  - To "inform" the operating system as it starts up (see sidebar)

Configuration files in Linux/Unix

- Configuration files are used to specify critical operational parameters such as:
  - where is the keyboard, what is the monitor resolution and scan rate etc.
- Configuration files make it possible to modify the OS for a given CPU configuration
- Early versions of Unix and Linux came as source code. By editing and configuring files and recompiling the OS, the OS could be easily customized to various target platforms.

Internationalization - I

- At design stage
  - we should consider internationalization issues
- At construction stage
  - we should implement internationalization, but solutions will vary according to the platform
    - E.g., Windows resource files (see sidebar)

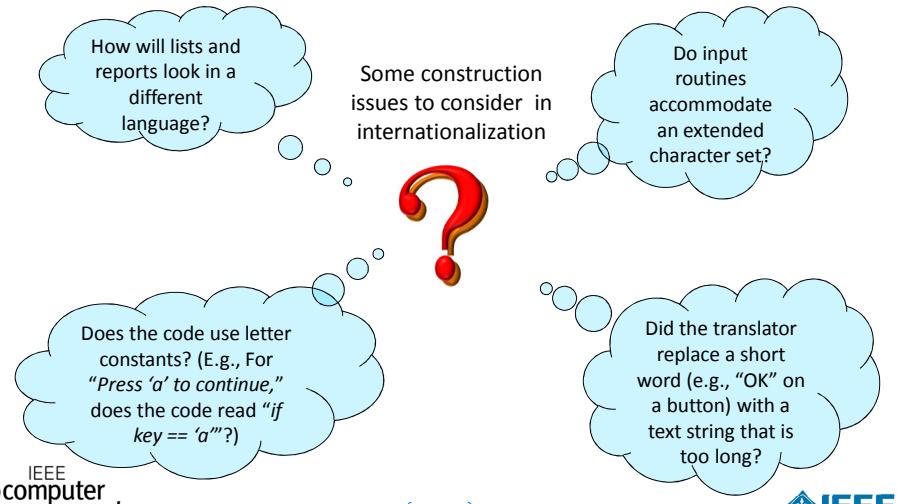
Some construction concerns

- Where to put text strings for messages, buttons, and menus that may need to be translated for the application to be useful in different countries
- How to handle variations in currency and calendar calculations

Internationalization in Windows applications

- Windows practice is to keep text in resource files that are loaded during program execution
- Text stored in resource files is relatively easy to translate; the application queries the OS what language to use at runtime

Internationalization – II



ref 1-120

Parsing

- *Parsing:* the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar
- *Parser*: Component of a compiler that does parsing
  - Parser transforms input text into a data structure, usually a tree, that is suitable for later processing and that captures the implied hierarchy of the input
  - Typically written using compiler-generator tools like YACC (Yet Another Compiler Compiler)

Why understanding parsing is necessary?

- Parsing used to be the domain of compiler writers using tools like YACC
- Parsing is now common place with wide use of HTML (HyperText Markup Language) and XML (eXtensible Markup Language)
  - Almost all browsers parse and execute HTML
  - XML is now widely used in industry for data representation and sharing; XML is processed using parsers

ref 1-121

Middleware

- Middleware is software that supports distributed applications
  - At a logical communication level, the software consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network
  - At a component level, middleware provides the basis for development of compatible components
- Standards such as CORBA and COM facilitate logical object communications on different platforms
  - They also provide a basis for implementing components with standard methods that can be queried and used by other components

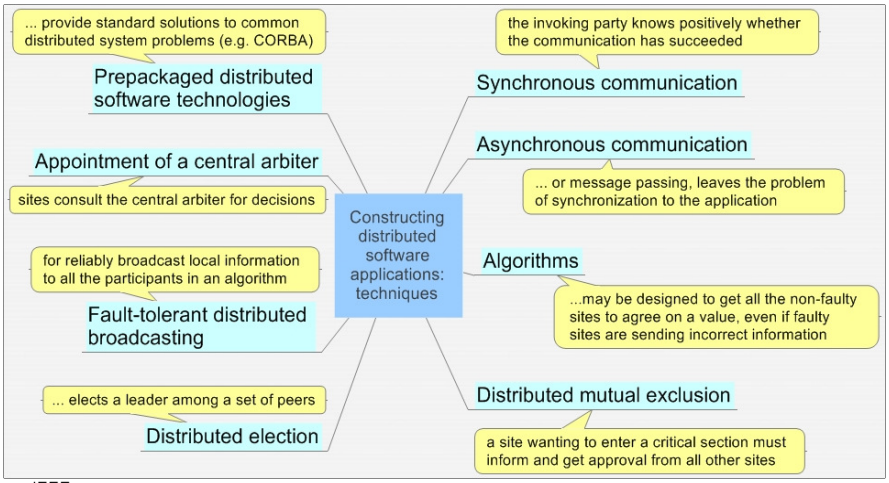
 [ref 1-121]

Distributed Software - I

- Distributed software
  - A system with two or more independent processing sites
- Distributed systems can be easy to modify and very reliable
  - However, construction of a distributed system faces some unique challenges
    - Under some circumstances, the transmission delays may exceed the time between state changes
    - The principal problem that distributed systems encounter is unpredictable, asynchronous failures. These may be related to the processing site, the communication media, or transmission delays
- Techniques for constructing distributed software applications
  - See next slide

 [ref 1-121]

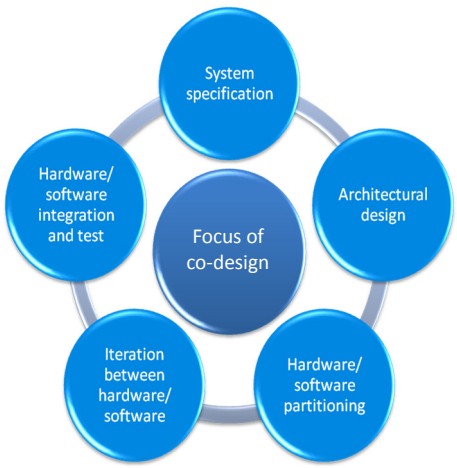
Distributed Software - II



 ref 1-122

Hardware/Software Co-design

- Co-design: simultaneous design of both hardware and software
  - E.g., cell phones, microwave ovens
  - focuses on system specification, architectural design, hardware/software partitioning, and iteration between hardware and software as design progresses
  - implemented by hardware/software integration and test
  - Four models are often used for co-design – see next slide

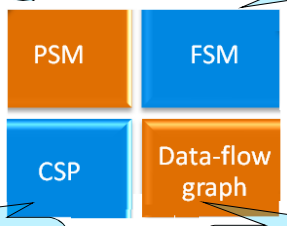


 ref 1-123

Models often used for co-design

Program State Machine model combines FSM and CSP

Finite State Machine (FSM): System represented as set of states; system transitions from state to state



Communicating Sequential Processes(CSP): System decomposed as concurrent processes and processes execute sequentially

decomposes functionality into data-transforming activities and the data flow between the activities

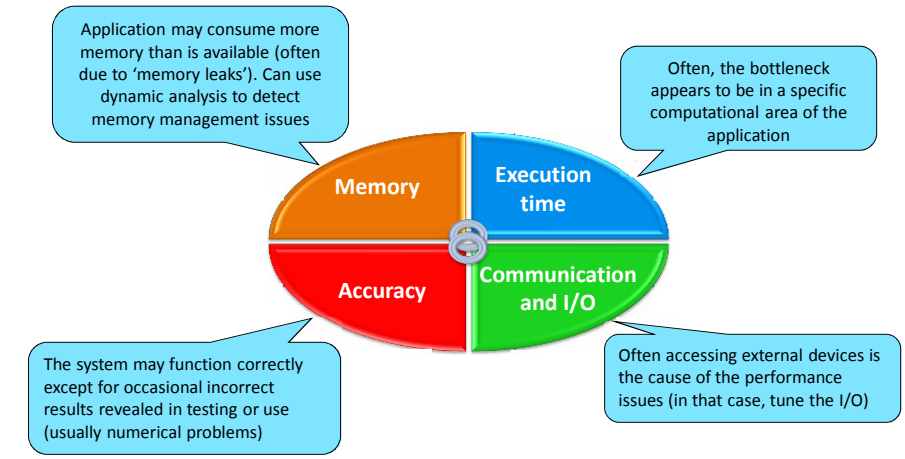
  ref 1-123

Performance Analysis and Tuning

- Performance issues often found during the test and integration process
  - Determine whether problem performance or perception
  - Determine whether problem is due to a difference between development/test environment and production environment
- The common performance problem indications:
  - slow response
  - seemingly random crashes
  - Insufficient throughput
  - intermittently incorrect results

  ref 1-123

Performance Problems: Common Causes



 ref 1-123

Performance Tuning – I

Tuning Memory Performance

- Use efficient representation
- Use only necessary memory
- Be sure the de-allocation function is working if dynamic memory is being used
- Replace complicated logic with static tables if possible
- Understand and reduce static memory requirements
- Use fewer objects and threads and keep them around longer

Improving Execution Time

- Compute values once and store them for subsequent use
- Simplify expressions
- Eliminate small loops and replace them with sequential statements
- Minimize the work done by a loop that is repeated many times
- Order case selectors according to their frequency or occurrence
- Consider converting critical sections to assembly language

 ref 1-123

Performance Tuning – II

Addressing Numerical Problems

- Calculating a value so close to zero that it gets rounded to zero
- A sequence of calculations that goes beyond machine accuracy somewhere in the middle. Analyze the compiled code to see how the operations are sequenced by the compiler
- The calculation may itself be incorrect: Confirm it is a development error and a requirements error

Tuning I/O

- Confirm that the protocols are understood and are being used correctly
- Measure the overhead of the communications mechanisms
- Measure the latency of the device (How long after it receives a request can it take action?)
- Determine if the device is shared or if exclusive access is available
- Determine the total load on the device and the communication channel

ref 1-123

Discussion Question

Which of the following would enable a programmer using C++ or Java to generate fault-tolerant, reusable interfaces that will save effort in the long run?

A. Putting exception handling as close to the point of the exception
B. Using assertions to document and verify pre- and post-conditions against use cases and requirements
C. Avoiding putting executable code in assertions
D. Using parameterized types or generics

Answer D: Using parameterized types or generics

Discussion Question

Which of the following is NOT a defensive programming technique?

A. Use named constants rather than hard-coding the numbers
B. Validating input from the user before processing the data
C. Including "self-checks" to confirm that the program is running properly
D. Using assertions to verify pre-conditions and post-conditions

Answer A: Use named constants rather than hard-coding the numbers

Rationale: Using named constants help in understanding and maintaining the program and is nothing to do with defensive programming; the other three are defensive programming techniques

Discussion Question

A software development team had just completed implementing an application. During system testing, it was found that the application was slow and did not meet performance requirements. Which of the following is the *most* appropriate action at this stage?

A. Change all looping constructs to improve efficiency
B. Replace recursion with iteration everywhere in the code
C. Use performance analysis tools to find and fix efficiency issues
D. Procure more powerful hardware/machine

… Discussion Question

Answer C: Use performance analysis tools to find and fix efficiency issues

Performance tuning is best done using performance monitoring tools that enable the developers to find sections of code that run slowly (“hotspots”) and improve them.

  ref 1-107

6. Product Documentation

Documentation

- During construction, documents prepared in previous phases are updated
  - Such as 'concept of operations', 'system requirements specification', 'software requirements specification', 'software design description' …
- Specific documentation depends on project & SLC model
  - Documents should be tailored according to relevant standards
    - *IEEE Std. 12207.1-1997: Software Life Cycle Processes - Life Cycle Data*
    - *IEEE Std. 829-1998: IEEE Standard for Test Documentation*
- Unit testing (and its documentation) takes place in construction phase

ref 1-124

Any Questions Regarding Module ?

Suggested Reading

- *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, 2004 ed., Los Alamitos, California: IEEE Computer Society Press, 2004.
- Sommerville, Ian. *Software Engineering*, 8th ed. New York: Addison-Wesley, 2007.
- McConnell, Steve. *Code Complete*, Microsoft Press, 2004.
- Thayer, Richard H., and Mark J. Christensen, eds. *Software Engineering, Volume 1: The Development Process*, 3rd ed., John Wiley, IEEE Computer Society Press, 2005.