



A Foundation for Object Orientation

Copyright © ESI Technology Corporation

This lesson lays a foundation for Object Orientation that will be used in ensuing lessons and workshop exercises.



Lesson Objectives

2

Upon completion of this lesson, the student should be able to:

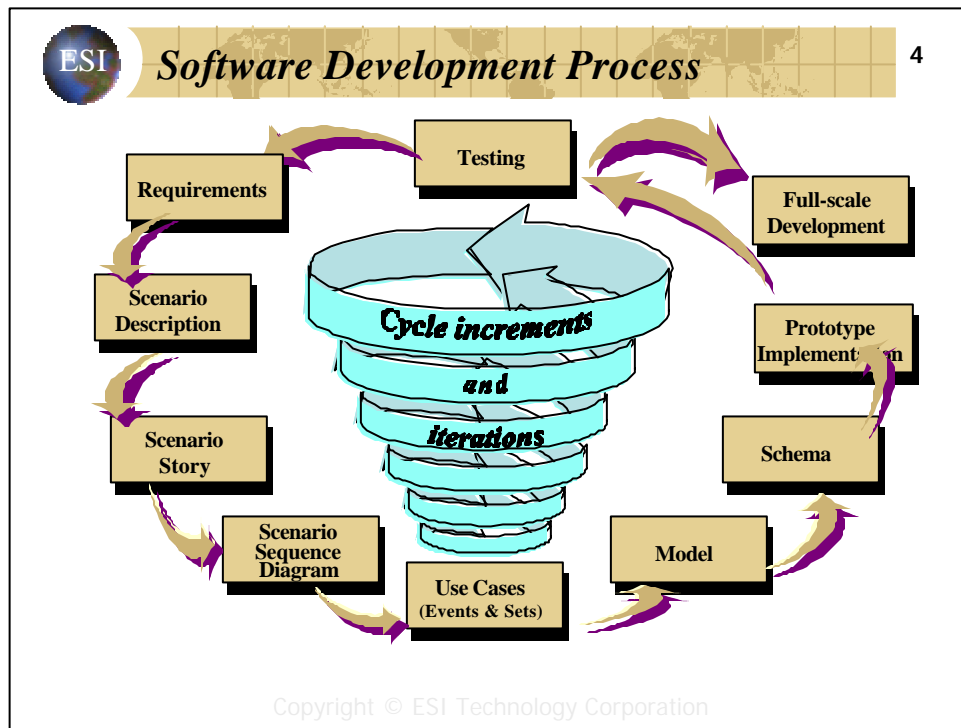
- Briefly introduce the modern day software development process.
- List and describe the software quality issues that effect the development of software systems today.
- List and describe criteria that can be applied to the software quality issues to produce good, sound software engineering principles.
- Describe and apply (throughout this tutorial) the principles of software engineering.

Copyright © ESI Technology Corporation

Read and understand the objectives of this lesson. We will refer back to these ideas as we explore object oriented concepts and more advanced features of the object orientation within the workshop exercises.



This lesson may seem academic to many, however, it lays a rigorous foundation for justifying the features of object orientation. Although, as software engineers, you should always apply the principles derived in this lesson. These principles should always serve as a guide in your engineering efforts. As a source of justification for object orientation, we will refer back to the principles as we go through future lessons and apply the concepts of object orientation.

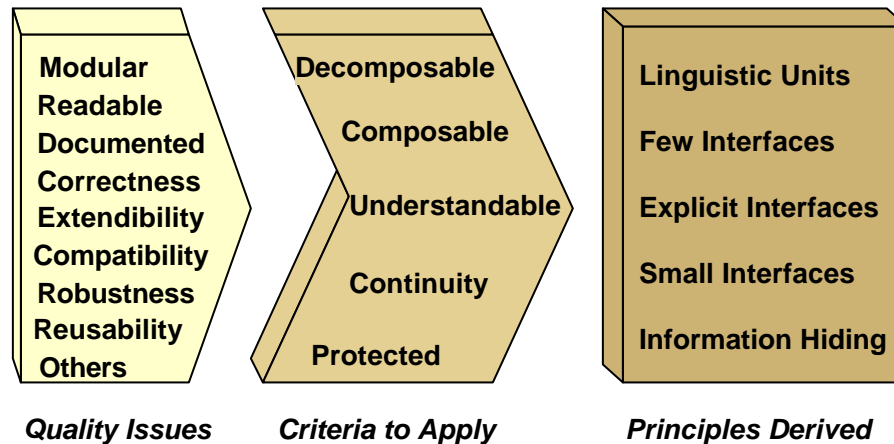


Although we will not be elaborating on the Software Development Process, it is important to understand it from a modern day perspective. The diagram above illustrates what is called the Iterative Process. Fundamentally, this approach makes a simple statement - software evolves! The old approach to developing software, commonly called the Waterfall Approach, has fallen by the wayside. Today software is developed in iterative stages - starting with a simple concept and evolving it to a final product rather than trying to create a final product in one step and then realizing it does not match the requirements.

As we will learn in this lesson, in order for the process to flow naturally, it is important that it all be founded on a common paradigm. That is, moving from one phase to the next should not require the engineer to change paradigms. This was the reason CASE failed in the 70's and 80's. There was no common paradigm linking each phase of process together.

As we go through the rest of the lesson, we will refer back to the diagram illustrated above.

For an excellent reference on the modern software development process refer to Jesse Liberty's book **Clouds to Code**. Additionally, for a complete description of Use Cases, read Ivar Jacobson's book **Object-Oriented Software Engineering**.



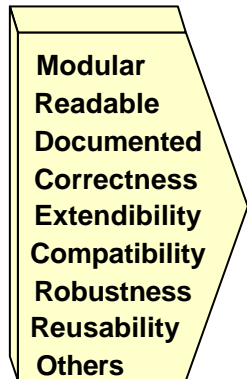
Object-Oriented Software Construction, Bertrand Meyer; Prentice Hall International Series in Computer Science

Copyright © ESI Technology Corporation

Throughout the history of software engineering, software quality issues have helped form the evolutionary foundation for the next generation of software. As engineering techniques evolved, so did the need to insure quality products. Today, the software industry is a multi-billion dollar industry. Software has invaded all aspects of our everyday lives. Consequently, quality issues have been cast into the lime-light. The Object Oriented paradigm grew out of the need to insure higher quality products.

Bertrand Meyer lays an air tight foundation for object orientation in his book **Object-Oriented Software Construction**. The diagram above illustrates his approach by starting with the most salient software quality issues. He then applies a set of criteria to those issues and derives a set of principles that identify the desired features of any software system.

The next few slides will briefly describe each of stages. It is not our intent to delve into these issues in depth, but to merely outline them in an axiomatic way. (We recommend reading Meyer's book if you are interested in the details.) However, the derivation process used by Meyer will be used throughout this tutorial to justify various aspects of the Object Oriented paradigm.



Quality Issues

Copyright © ESI Technology Corporation

Software quality issues have manifested themselves over the evolutionary life of software engineering. These issues are at the foundation of good software engineering principles. They can be viewed as internal and external.

INTERNAL

Readable - Can the engineer easily read and understand the code?

Modular - Is the software modular and well structured?

Documented - Is the software well documented, on-line and off-line?

EXTERNAL

Correctness - Does the software perform tasks according to the requirements and specifications?

Extendibility - Does the software adapt to changes in specification or requirements?

Compatibility - Can the software products may be combined with others easily?

Robustness - Can the software function properly under abnormal conditions?

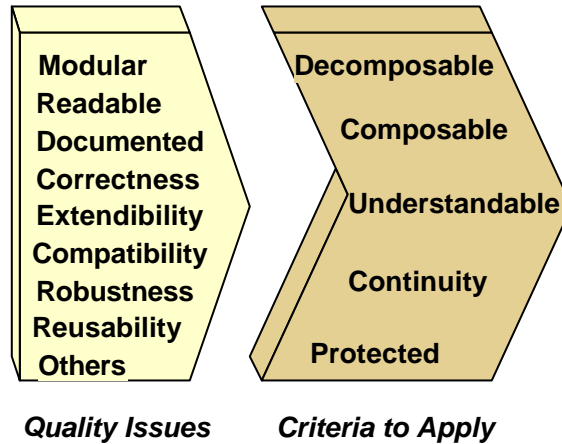
Reusability - Can the software to be reused, in whole or in part, in new applications?

Others - Is the software efficiency, verifiable, easy-to-use, portability, of high integrity, etc.



Criteria to Apply to Quality Issues

7



Copyright © ESI Technology Corporation

To derive the software quality principles, Meyer applies a set of criteria to the Quality Issues.

Decomposability - Can the problem domain can be broken into sub-problems?

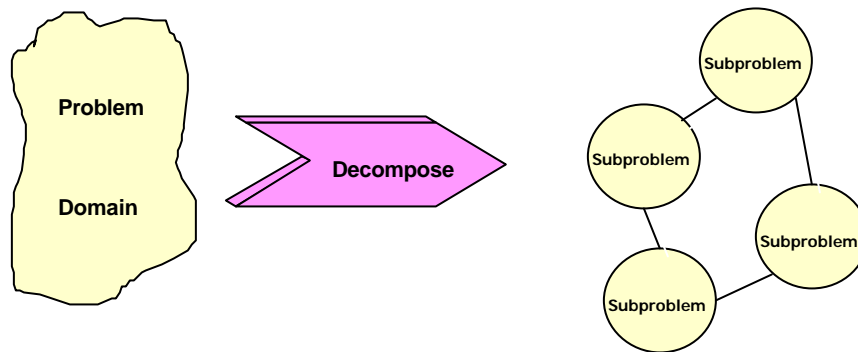
Composability - Can sub-problem domains be mapped into software modules?

Understandability - Can any one module be looked at in isolation and understood?

Protection - Is an anomaly at run-time confined to the module?

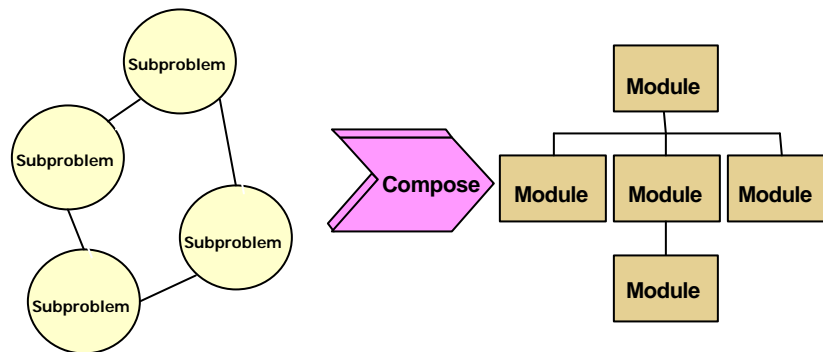
Continuity - Can a small specification change can be directed to a module without changing the architecture.

After listing the criteria, we can then extract a set of principles by applying the criteria to the quality issues. These extracted principles can then be applied to the software process. The next five slides will elaborate upon these criteria slightly.



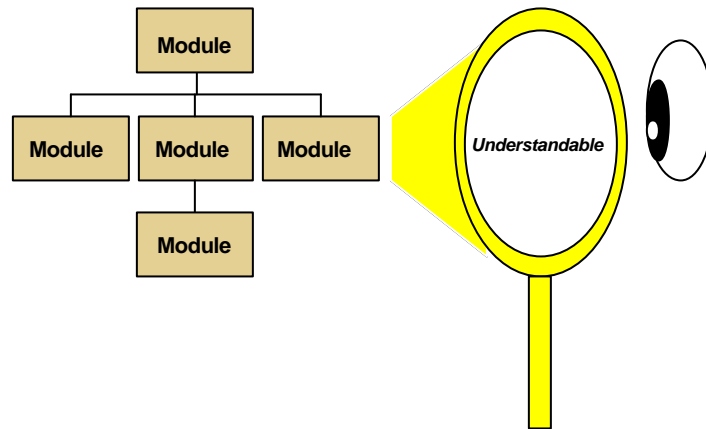
The **Decomposability Criteria**, when applied to the problem domain, should produce subproblem domains. This is a fundamental, prerequisite step to developing modules.

Within the object oriented software development process, it is important to develop scenarios consisting of one or more use cases. A use case is simply a description of how a user would perform a function within the proposed application. Collectively, the use cases identify all the objects, attributes, as well as responsibilities and collaboration between the objects. It is this process that decomposes the problem domain into sub-problem domains.



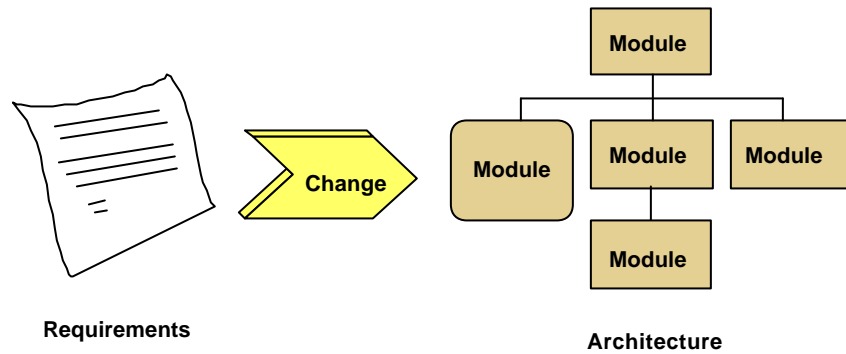
The **Composability Criteria**, when applied, should transform the subproblem domain into implementation module structures. Please note that the term module is used in a general sense, it could refer to a routine or an object.

Use cases should identify all the objects that will collaborate within the applications. They should also identify how these objects collaborate with each other and what their respective responsibilities are. Transforming (composing) these objects, their attributes and behavior into an object diagram is the first step to developing software modules. Tools exist today to assist in that process.



The **Understandability Criteria**, when applied to a module, means the module should make sense viewed by itself. In other words, it should have a well defined sense of its responsibilities and not be totally dependent upon other modules to do its job. Responsibility boundaries should be clearly established.

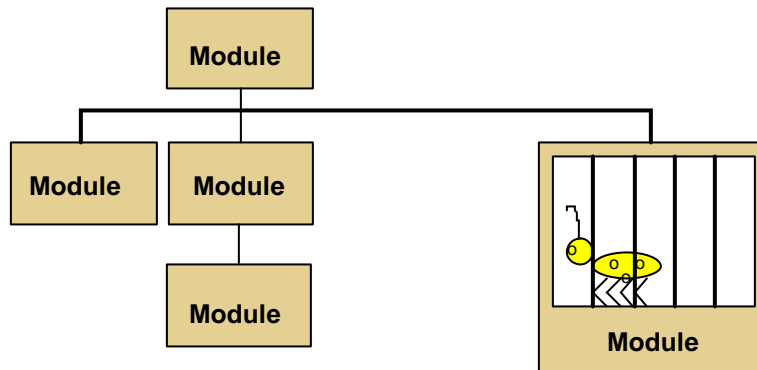
Within the object oriented development process, once modules have been designed, a walk-through should be conducted to make sure the objects display the proper responsibilities and collaborate with each other according to the use cases.



The **Continuity Criteria**, when applied, insures that when a change is made to the specification, the architecture will absorb that change with minimal impact upon the reset of the system. In other words, it is easily extensible and will not require a redesign to accommodate the change.

By applying the iterative approach and making sure that the previous criteria are adhered to, the continuity criteria can be applied with confidence.

Reusability is an important promise of object orientation. Continuity insures the fulfillment of that promise.



Copyright © ESI Technology Corporation

The **Protection Criteria** means that when an anomaly occurs, it is confined to that module and does not affect other modules. It does not produce the infamous 'ripple effect'. It's when modules are tightly bound and do not have a clear delineation of responsibility that we witness the problem.

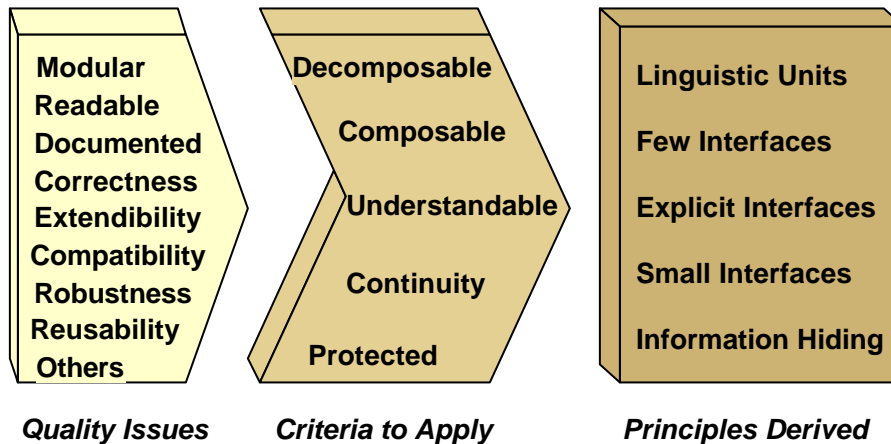
Enforcement of this criteria is difficult in traditional procedural systems. However, it is a natural consequence of proper application of the object-oriented paradigm WHEN encapsulation is enforced. Encapsulation will be elaborated upon in the next lesson. It is a simple concept. It simply states that an object's data and code are encapsulated and only accessible (by other objects) via a well-defined, public interface.

It is a fact that when encapsulation is enforced, bugs remain confined to their respective modules (objects!)



Derived Principles from Applied Criteria

13



Copyright © ESI Technology Corporation

The following principles are derived from the software quality issues when the criteria is applied:

Linguistic Modular Units - The formalism used to express the system to the outside world, as well as to the computer, should be the same throughout phases of the development process.

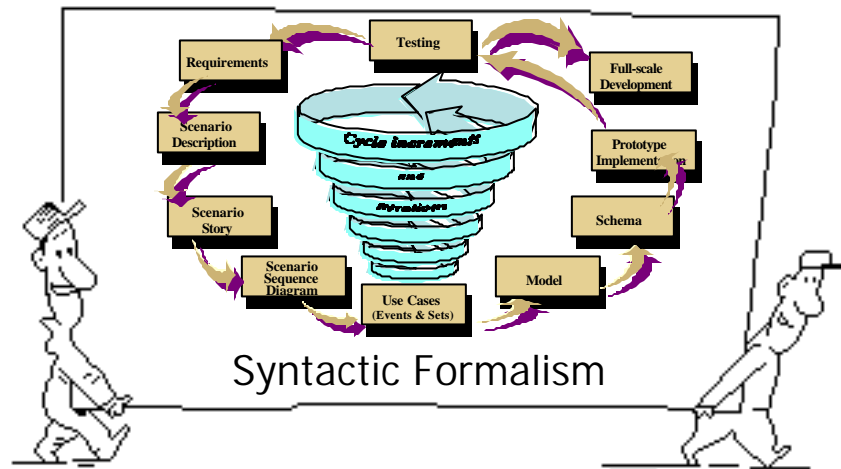
Few Interfaces - The number of communication channels or entry points of a module should be minimized.

Explicit Interfaces - Communications between two modules should clearly indicate what is happening. The interface structure should be common between modules.

Small Interfaces - Keep the amount of information transferred between modules small.

Information Hiding - All information about a module should be private to the module unless declared public.

The next few slides will elaborate upon these principles.

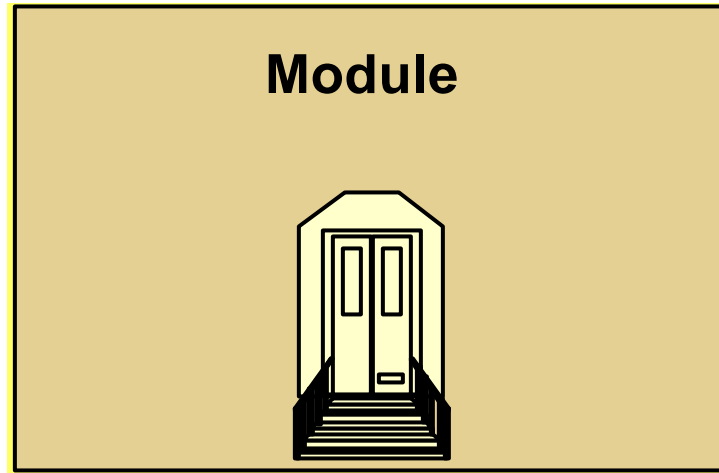


Copyright © ESI Technology Corporation

The **Linguistic Modular Units** principle states that within all phases of the software engineering process, the syntactic formalism used to express the requirements, specification, design and implementation should be based upon a common paradigm.

Object Orientation provides this formalism. Design specifications can be expressed in terms of objects. Designs can be created today using object oriented tools such as Rational Rose™. If the design consist of objects, properties, relationships, etc., then the language and development tools should permit implementation of the model without any hassle.

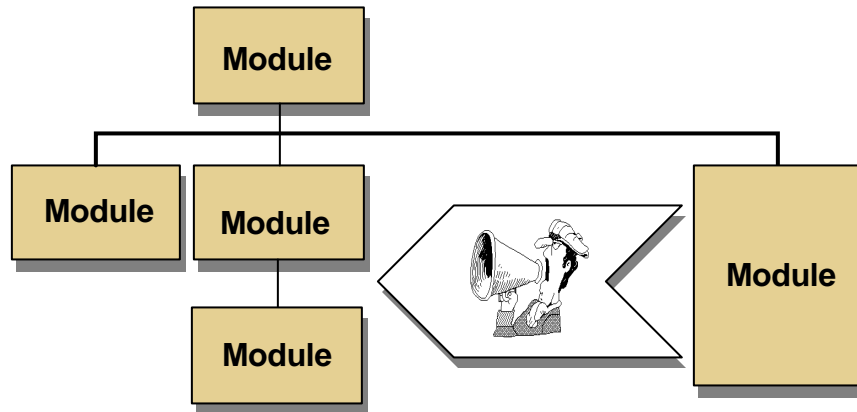
If the object paradigm provides the common thread through each phase, moving from one phase to another is a smooth process. The impedance mismatch commonly experienced by old CASE tools disappears.



Copyright © ESI Technology Corporation

The **Few Interfaces** principle states that the number of entry points into a module should be absolutely minimized, preferably one. Multiple entry points to a module can cause problems and subsequent integrity problems.

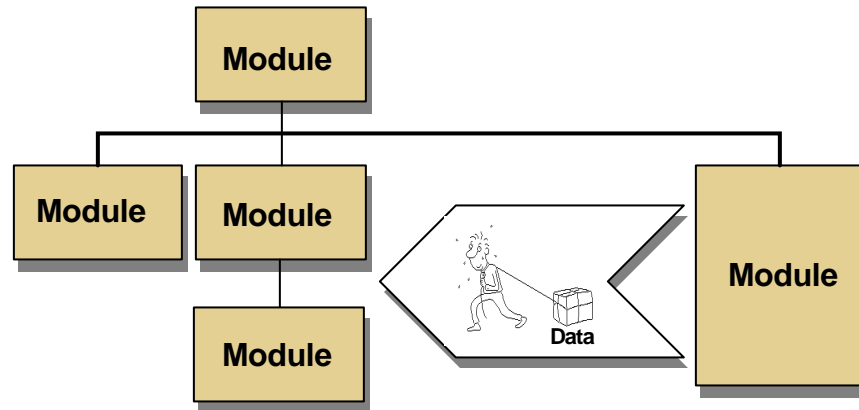
In good object oriented systems, an object usually will have one public entry point. Although the content will certainly differ, the structure of the entry point should be consistent and common to all objects within that system.



Copyright © ESI Technology Corporation

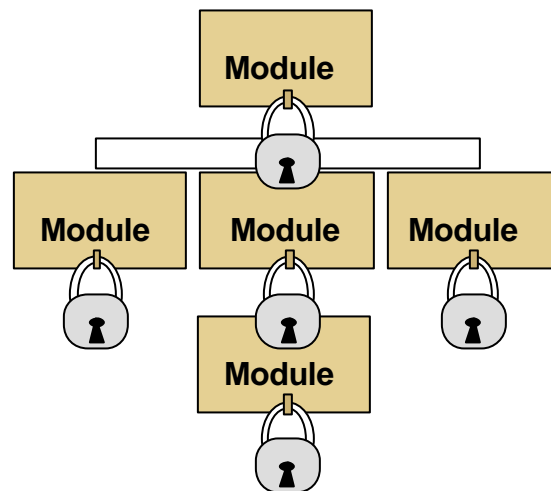
The **Explicit Interface** principle states that a modules interface should be well defined and unambiguous. Its structure should be common to all modules. This facilitates communications between modules.

A good object model, if implemented consistently, provides a common, consistent interface to objects in most commercial systems. All objects should obey the same protocol.



Copyright © ESI Technology Corporation

The **Small Interfaces** principle states that a module should not have to take in a whole lot of information to perform its responsibility. Generally, if a module must have inordinate amounts of information to operate properly, the module's responsibilities have been improperly allocated and it may be time to review the design. In object orientation, a small interface generally means the design is optimal and that responsibilities have been properly allocated. If interfaces are small, this usually has an impact on performance.



The **Information Hiding** principle states that all information about a module, or information the module is responsible for, should be private to that module unless it is declared as public.

The Information Hiding principle is fundamental to the Object Oriented paradigm. It manifests itself through the concept of encapsulation. Encapsulation will be elaborated upon during the OO Concepts lesson.



Lesson Summary

19



Copyright © ESI Technology Corporation

In summary, software quality issues have arisen out of the historical evolution of software. As we evolved from writing simple linear programs to highly complex systems that required more structured development techniques and tools, we evolved some common software engineering principles. The principles outlined in this lesson are universal to all kinds of software engineering. However, the object oriented paradigm is clearly based upon these principles as we will see in the next lesson.



End of Lesson - What's Next?

20



Copyright © ESI Technology Corporation

The next lesson (Lesson 3) will introduce you to Object Oriented Concepts. It attempts to lay a solid conceptual foundation to build upon as you begin your object oriented programming career. Concepts learned in the next lesson are universal and independent of any specific OO implementation. They are concepts that are considered essential to a good OO system.

Proceed to the next lesson [Object Oriented Concepts](#) once it is published.