**Module 7**

Software Design Concepts:

* Abstraction
* Architecture
* Patterns
* Separation of Concerns
* Modularity
* Information Hiding
* Functional Independence
* Stepwise Refinement
* Refactoring
* Design Classes

Abstraction

* process – extracting essential details
* entity – a model or focused representation

Information Hiding

* the suppression of inessential information

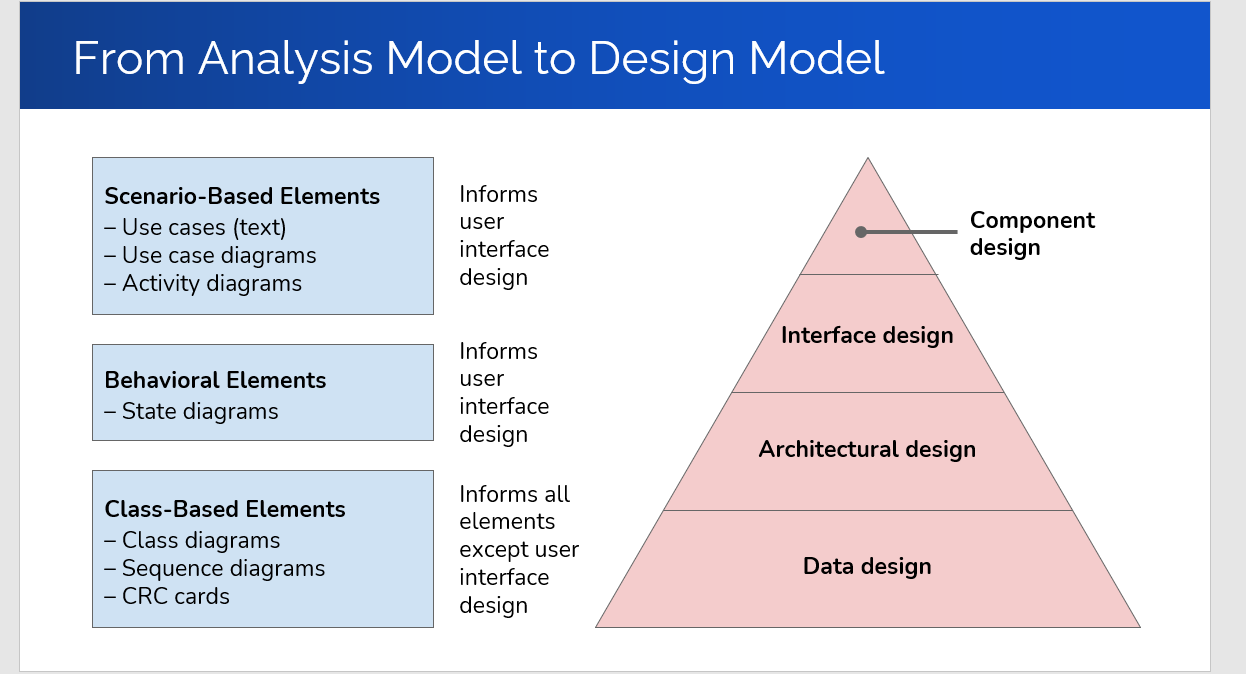
Encapsulation

* process – enclosing items in a container
* entity – enclosure that holds the items
* **Cohesion** – the degree to which a module performs one and only one function.
* **Coupling** – the degree to which a module is connected to other modules in the system.

**Refactoring** is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

When software is refactored, the existing design is examined for:

* redundancy
* unused design elements
* inefficient or unnecessary algorithms
* poorly constructed or inappropriate data structures
* any other design failure that can be improved upon



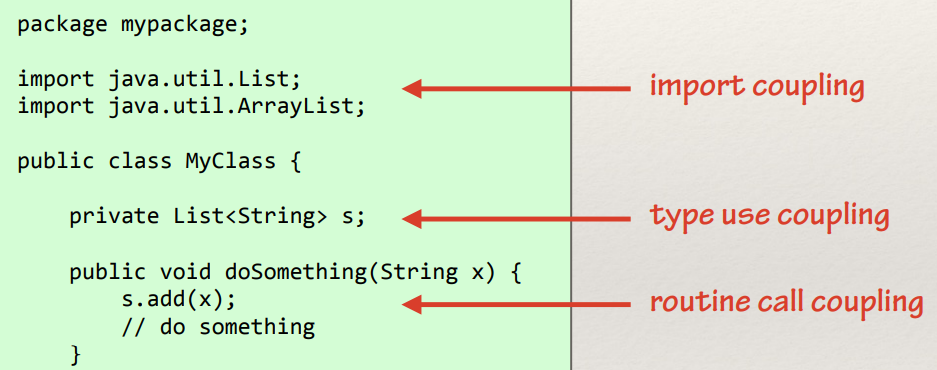
* Component: A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces – OMG UML Specification
* OO View – A component is a set of collaborating classes.
* Conventional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.
* Symptoms of Rotting Design:
  + Rigidity – tendency for software to be difficult to change
  + Fragility – tendency for software to break when it is changed
  + Immobility – inability to reuse software from other projects
  + Viscosity – easier to hack software than to keep design
* **Software rot** is the result of improper dependencies between modules
* Dependency among modules must be managed – use design principles
* Principles of OO Class Design:
  + Open-Closed Principle
  + Liskov Substitution Principle
  + Dependency Inversion Principle
  + Interface Segregation Principle
* Open-closed: A module should be open for extension but closed for modification. (In other words, we want to be able to change what the modules do, without changing the source code of the modules)
* Abstraction is the key to the OCP.
* Design by Contract: The relationship between a class and its clients can be viewed as a formal agreement, expressing each party’s rights and obligations
* A precondition is a statement of how we expect the world to be before we execute an operation.
* A **robust** program is one that continues to behave reasonably even in the presence of errors. A program like this is said to provide graceful degradation
* Liskov Substitution Principle: Subclasses should be substitutable for their base classes. (A user of a base class should continue to function properly if a derivative of that base class is passed to it.)
* In terms of contracts, a derived class is substitutable for its base class if

1. Its preconditions are no stronger than the base class method

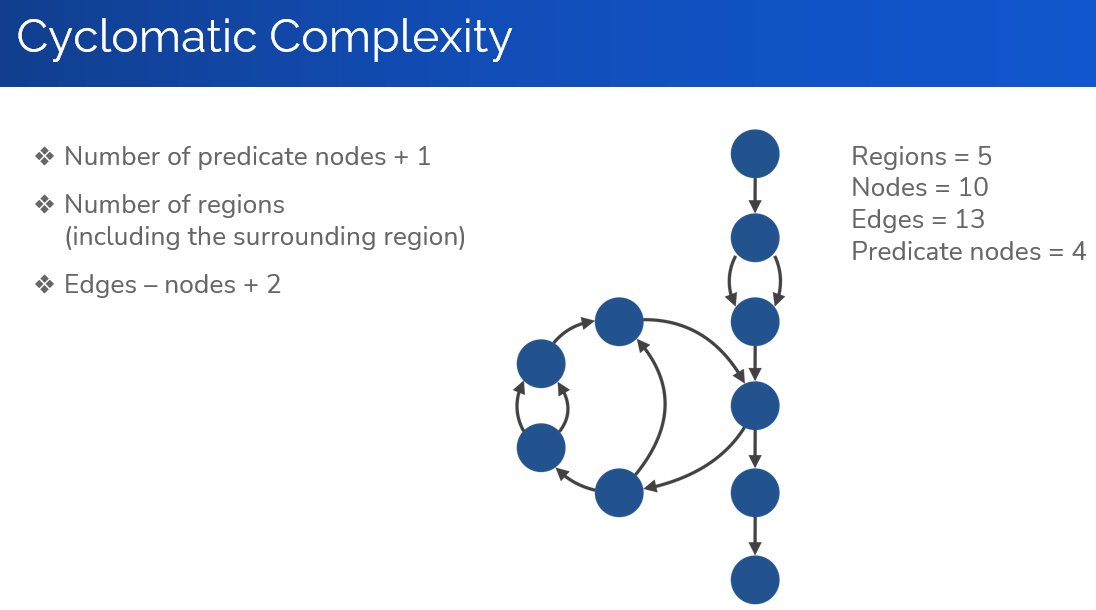
2. Its postconditions are no weaker than the base class method

❖ In other words, derived methods should expect no more and provide no less

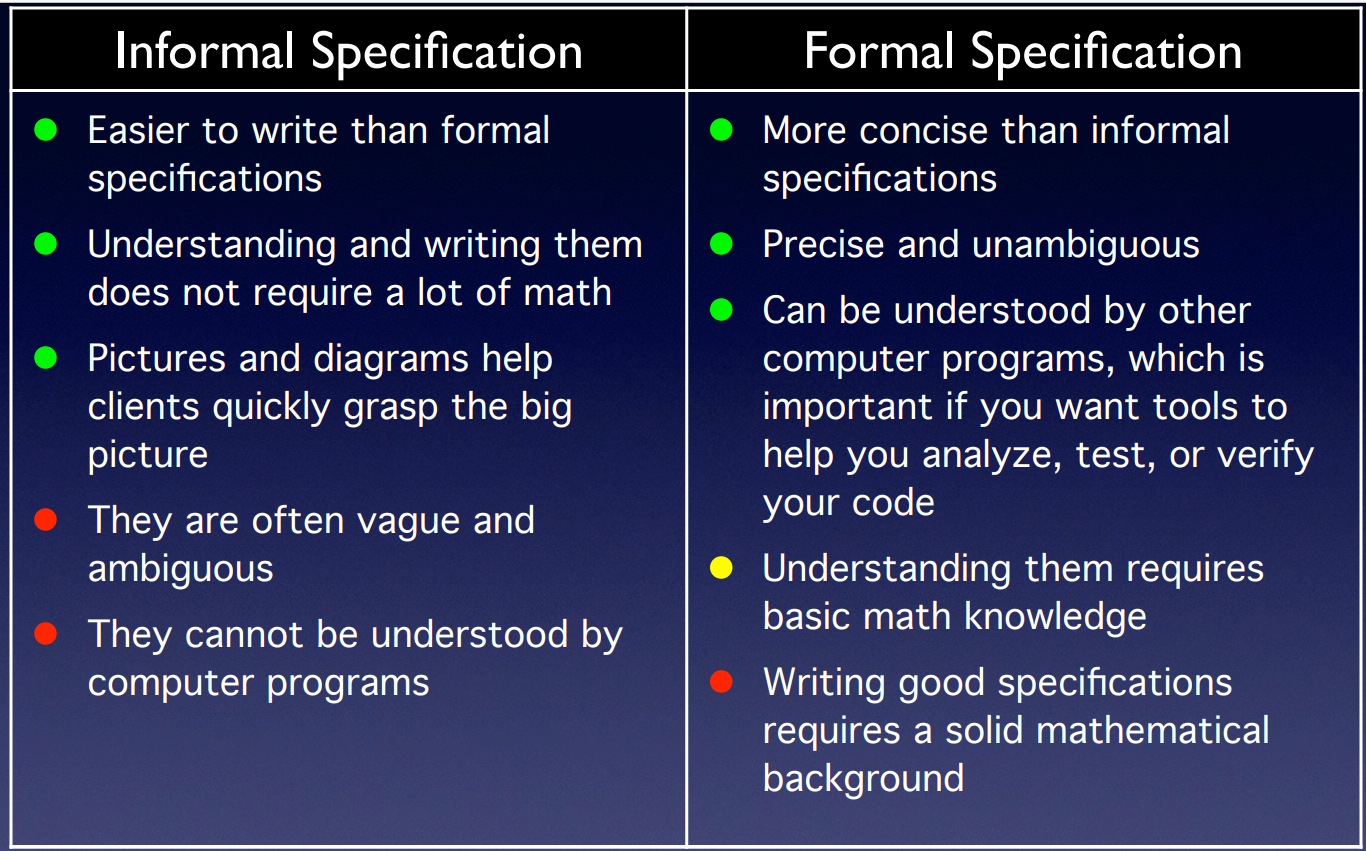
* Dependency Inversion Principle: Depend upon abstractions. Do not depend upon concretions.
  + Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.
* A component should be usable solely on the basis of its specification.
* Modular (Component) Reasoning: Different distributors may be used in the same engine as long as they meet the same (physical) specification
* Interface Segregation Principle: Many client specific interfaces are better than one general purpose interface.
  + If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.
* Package Cohesion Principles:
  + The Release Reuse Equivalency Principle
    - The granule of reuse is the granule of release
  + The Common Closure Principle
    - Classes that change together, belong together
  + The Common Reuse Principle
    - Classes that aren’t reused together should not be grouped together.
  + The Acyclic Dependencies Principle
    - The dependencies between packages must not form cycles
  + The Stable Dependencies Principle
    - Depend in the direction of stability
  + The Stable Abstractions Principle
    - Stable packages should be abstract packages.
* Cohesion:
  + The “single-mindedness of a module”
  + A component should contain only attributes and operations that are closely related to one another and to the component itself.
  + Types of cohesion include:
    - Functional: A function should perform only one task
    - Layer: A higher layer in a system can access a lower layer, but not the other way around
    - Communicational:
      * All operations that access the same data are defined within one component.
      * Often, such components focus solely on the data in question, accessing and storing it.
      * Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components.
* Coupling:
  + A qualitative measure of the degree to which classes or components are connected to each other.
    - content coupling – **avoid**
      * Occurs when one component “surreptitiously modifies data that is internal to another component”
      * Violates information hiding
    - common coupling – **use** **caution**
      * Components use the same global variable
    - routine call coupling, type-use coupling, import coupling – **be aware of**
      * Some types of coupling occur routinely in object-oriented code



**Module 10**

* Software testing: Software is tested to uncover errors that were made inadvertently as it was designed and constructed.
* **The developer:** Understands the system, but will test gently and driven by delivery
* **Independent tester:** Must learn about the system, will attempt to break it, and is driven by quality.
* **Verification:**
  + Are we building the product right?
  + Is the code correct with respect to the specification?
  + Is construction consistent with modeling?
* **Validation**
  + Are we building the right product?
  + Does the specification reflect what it should?
  + Is the model consistent with the intended requirements?
* Unit -> Integration -> Validation -> System
* **Unit Testing:** Tests individual components
* **Integration Testing:** Tests how components interact with one another
* **Regression Testing**: The selective retesting of a modified system to help ensure that no bugs have been introduced during modification. (Changing one part of the code can break another.)
* **Validation Testing:** The focus is on software requirements
* **System Testing:** The focus is on system integration
  + **Alpha/Beta testing** – The focus is on customer usage
  + **Recovery testing** – Forces the software to fail and verifies that recovery is properly performed
  + **Security testing** – Verifies that system is protected from improper penetration
  + **Stress testing** – Executes system so that it demands resources in abnormal quantity, frequency, or volume
  + **Performance testing** – Tests run-time performance
* **Black-Box Testing:** Treat the component as a black box (the implementation is hidden)
* **White-Box Testing:** Ensure that each statement in the implementation is executed at least once.
* Although black box testing is sometimes referred to as specification-based testing, test cases in black box and white box testing are both based on the specification – the output you expect is the output the specification says you will get.
* Black-Box:
  + Treat the component as a black box and develop your test cases based only on information given in the specification.
  + Partition the input and output around sensible boundary points. Test each partition and test at or near each boundary point.
* White-Box:
  + Guarantee all paths in a module are executed at least once
  + Exercise all conditions on both true and false sides
  + Exercise all loops at boundaries and within boundaries
  + Exercise internal data structures to ensure their validity
* Path Coverage:
  + Multiple paths are created by conditions, found in if-statements, while-loops, and so on...
  + A program with no conditions has only one path
  + Covering every path is known as **exhaustive testing**
  + Exhaustive testing is not practical
  + Even if exhaustive testing is achieved, you can still have incorrect code
* 
* It is **NOT** possible to cover all linearly independent paths using just one test case

**Module 11**

* A component should be usable solely on the basis of its specification
* Specification: What it does
* Implementation: How it does
* Informal Spec: Describe what a component does using natural language, pictures, or real-world metaphors
* Formal Spec: Describe what a component does using a mathematical specification language
* 
* **Grand Challenge for Computing Research**
  + The construction and application of a **verifying compiler** that guarantees correctness of a program before running it. - Tony Hoare, 2003
* Typical Grand Challenges:
  + Prove Fermat’s last theorem (done)
  + Put a man on the moon (done)
  + Cure cancer in 10 years (failed in 1970’s)
  + Prove that P is not equal to NP (open)
  + Turing test (done)
  + Championship chess program (done)