Lecture 15

The Object-Oriented Design Principles

Nature Creates and Maintains with Maximum Efficiency

Wholeness Statement

The most important software engineering design principles are encapsulation, loose coupling, high cohesion, separation of concerns, and subtyping. Careful adherence to OO and software engineering design principles create systems that are flexible, reusable, self-contained/modular, and maintainable. *Science of Consciousness:* Creation of any object in creation is ultimately the result of the dynamic natural laws (principles and algorithms) of the unified field of pure creative intelligence. Experience of this field brings the positive qualities of this field into the life of an individual and society as demonstrated by many scientific research studies.

Important Concepts in OO

- **Fundamental Software Engineering Principles:**
  - Encapsulation
  - Loose Coupling
  - High Cohesion
  - Separation of Concerns
  - Subtyping & substitution
- **Fundamental OO Concepts:**
  - Class
  - Object (Instance)
  - Attribute (State)
  - Method (Behavior) & Messages
  - Program to Interface not the Implementation
- **Key OO Concepts:**
  - Instantiation
  - Abstraction
  - Implementation Hiding
  - Composition
  - Inheritance
  - Polymorphism

Important Principles in OO Design

- Reuse
- Collaboration, Composition, & Delegation
- Favor Composition & Delegation Over Inheritance
- Open/Closed Principle
- Substitution Principle

Classes & Objects

- **Objects** have:
  - **State**
    - Represented through ***attributes*** (fields)
    - “Local” variables applicable only to a single object
  - **Behavior**
    - Provided through ***methods***
      - A procedure with a meaningful side effect
        - Like modifying attributes of an object
        - Or printing or persisting (saving) data
      - Or a function that performs computations and returns a value
  - **Identity**
    - So objects can be distinguished even if they have the same state and behavior
- **Classes**
  - Define **properties** which are mappings from names to ***attributes*** and ***methods***

Abstraction

- Removes unnecessary details
  - Allows us to ignore details and to treat things that are different as if they were the same (e.g., ArrayList vs. LinkedList are both a List)
  - For example, programming in a high-level language eliminates the low-level details of the machine instructions generated by the compiler
- Not really practical to build all needed abstractions into a programming language
  - Would make the language difficult to learn or use
- The better approach
  - Build into the language the necessary mechanisms so programmers can create their own abstractions
  - Create a library of standard abstractions

Kinds of Abstractions (Liskov)

- Procedural abstraction
  - Ability to add/create new operations (functions)
- Data abstraction
  - Ability to create new data types
- Type hierarchy
  - Ability to create a family of related data abstractions
- Iteration abstraction
  - Ability to iterate over a collection of objects without revealing how they are stored or retrieved

Procedural Abstraction

- Separates definition from invocation
- Allows two kinds of abstraction
  - Abstraction by parameterization
    - Separates data from the algorithm
    - Example: Creation of parameterized methods/functions or generic classes
  - Abstraction by specification
    - Separates expected behavior (what) from implementation (how)
    - Example: creation of specifications for users of a class library

Abstraction by Parameterization

- The data is passed in as parameters
  - Separates input data from the algorithm
- Generic types do the same for data abstractions
  - The parameters are types rather than data since a type is static (like a blue print)
  - Not applicable in JavaScript since there are no declared types

Abstraction by Specification

- Abstracts away the implementation details
- Specifies the behavior the user can depend on
  - Can be formal (Java Modeling Language) or informal (Java docs)
- Only requirement is that the implementation must satisfy the specification

What is the difference between a subclass and a subtype?

Subtypes

- Subtypes must satisfy the substitution principle
  - Subclasses must not cause supertypes to behave differently from their specification
  - That is, clients must be able to write and reason about their code from the supertype specification anytime the code references a subtype
- Properties of a subtype
  - Must have all the methods of its supertype
  - Overriding methods must “behave like” the overridden superclass method
    - (requires a superclass specification)
  - Must preserve all invariant properties of the supertype

Example of a Subclass of the JS Template Method EulerTour

```
class EulerTour {
    visitExternal(T, p, result) { }
    visitPreOrder(T, p, result) { }
    visitInOrder(T, p, result) { }
    visitPostOrder(T, p, result) { }
    eulerTour(T, p) {
        let result = new Array(3);
        if (T.isExternal(p)) {
                this. visitExternal(T, p, result);
        } else {
                this.visitPreOrder(T, p, result);
                result[0] = this.eulerTour(T, T.leftChild(p));
                this.visitInOrder(T, p, result);
                result[2] = eulerTour(T.rightChild(p));
                this.visitPostOrder(T, p, result);
        }

        return result[1];

}
```

```
public class Sum extends EulerTour {
    sum(T) {
        return this.eulerTour(T, T.root());
    }

    visitExternal(T, p, result) {
        result[1] = 0;
    }
    visitPostOrder(T, p, result) {
        result[1] = result[0] + result[2] +
                    p.element();
    }

    …
}
```

Subclassing Example

```
class BinaryTree {
}
class BinarySearchTree extends BinaryTree {
}
```

- Has the required isA relationship
- Problem is that there are methods of the BinaryTree that we do not want to be invoked by clients of BinarySearchTree
  - For example, insertLeft(p,e) of BinaryTree
  - Thus BST should be implemented as having a BinaryTree attribute and delegate operations to the internal BinaryTree

Subtyping Example

```
class BinarySearchTree extends BinaryTree {
}
class RedBlackTree extends BinarySearchTree {
}
```

- Here we can allow every method of BST to be inherited or overridden by RedBlackTree
- So it is a subtype, i.e., a RedBlackTree can be substituted anywhere a BST is used since all methods can be invoked
- Therefore, RedBlackTree is a subtype of BST since every method of subclass behaves like the superclass
  - This was not true for some methods of BinaryTree since they could ruin the ordering properties of the BST

Main Point 1

In OOP, abstraction is realized by combining data and behavior into an object. Subtypes are created by defining object types that behave like and are substitutable for their supertype objects.

Science of Consciousness: In enlightenment, one “behaves like natural law”, i.e., behavior is spontaneously life-supporting and in full accord with natural law for the benefit of everything and everyone.

Class Design through Software Engineering Principles

Encapsulation,
Coupling, Cohesion, Separation of Concerns, & Information Hiding

Encapsulation, Coupling & Cohesion

- Encapsulation
  - Grouping of related items or concepts into one item, such as a class or component (package, namespace)
- Coupling
  - The degree of dependence between two items
- Cohesion
  - The degree of relatedness of the elements of an encapsulated unit

Class Design

- A class is an ***abstraction*** of a real world object or concept
- ***Encapsulates*** data and operations
- ***Hides the implementation*** of its attributes and methods
- Attributes and methods should have ***high cohesion*** and ***loose coupling***

Encapsulation

- Deals with
  - how to modularize the features of a system
  - how functionality is compartmentalized within a system
- In OO, a system is modularized into classes, which are modularized into methods and attributes (fields)
  - behavior is encapsulated into a class
  - functionality is encapsulated into a method

Implication of Encapsulation

- We can later change the implementation without affecting other components of the system
  - As long as the interface and behavior of that component does not change
- We can enforce this through <u>information hiding</u>
  - Examples are private attributes (fields) and methods (in JavaScript we prefix the names with a "_" to indicate that this method is not to be overridden or that an attribute/field is not to be accessed outside of the methods of the enclosing class
  - If a method is <u>specific</u> to the implementation, then it should be private or at least protected in Java (or prefixed with a "_" to indicate this)

Coupling

- Measures how two items, such as classes and methods, are interrelated
- Two classes are ***coupled*** if one class depends on the other
- Two classes are ***loosely coupled*** if they interact but neither class knows (depends on) any of the implementation details of the other
- Two classes are ***tightly coupled*** if one class relies on the implementation of the other
  - it directly accesses the data attributes of the other
  - it depends on how a method is implemented (e.g., which methods are called or the types of internal, non-public fields of the class)
  - if two classes are tightly coupled, a change in one often requires a change in the other
- Coupling of methods is measured in the same way, i.e., whether one in any way depends on the implementation of the other

Example of Tight Coupling

- The programmer of Student class knows that the listOfStudents attribute in Seminar class is implemented as an array
- Then he/she decides to add a Student object to the listOfStudents inside the Student class by directly accessing the Seminar's listOfStudents array

Example of Tight Coupling

```
class Seminar {
    constructor () {
        this._listOfStudents = [ ];
        this._studentCount = 0;
    }
    register(student) {
    }
}
class Student {
    register(sem) {
          sem._listOfStudents[_studentCount] = this;
          sem._studentCount ++;
   }
}
```

Example

- Later the listOfStudents implementation is changed from an array to a linked list

```
class Seminar {
    constructor () {
            this._listOfStudents = new List();
            this._studentCount = 0;
    }
    register(student) {
    }
}
class Student {
    register(sem) {
        sem._listOfStudents[_studentCount] = this;
        sem._studentCount ++;
   }
}
```

- **If the Student class is not changed, the system will crash!**

Example (Correct way)

- Later the listOfStudents implementation is changed from an array to a linked list

```
class Seminar {
      constructor () {
            this._listOfStudents = new List();
            this._studentCount = 0;
      }
      register(student) {
            …
      }
}
class Student {
      register(sem) {
         sem.register(this);
      }
}
```

- **The Student class does not need to be changed!**

Example

- If access to listOfStudents had been restricted, i.e., not directly accessed
  - Programmer of Student would have been required to ask Seminar objects to add a Student object to the list
  - By hiding the implementation (the seminar list) and encapsulating how students are enrolled in courses, we are able to keep the abstraction the same while allowing the implementation to possibly change
- Principle:
  - Program to the interface (specifications of the methods), not to the implementation of the data and methods

Tight Coupling

- Tight coupling only makes sense when you are truly desperate to cut down on the processing overhead
  - E.g. database drivers are often tightly coupled to the file system of the operating system on which the database runs
- In the Java compiler that I worked on, we used JavaDocs heavily as part of our documentation, but every time Java released an updated version, our system failed because of tight coupling in the implementation and changes in the interfaces

Implementation Hiding

- Basic idea:
  - If one class wants information about another class, it should have to ask, instead of directly accessing it
  - This is how the real world works:
    - If you want to know someone’s name, you ask them
    - You don’t steal their wallet
- Makes an application easier to maintain

Implementation Hiding

- Prevents other programmers from writing highly coupled code
  - When code is highly coupled, a change in one part of the code forces changes in another, and then another, and so on.
- It is up to us how we implement something and we should be able to change the way we implement it at any time:
  - This is possible through Encapsulation
  - Encapsulation absolutely requires Implementation Hiding
    - Encapsulation means we should be able to change the implementation without affecting other classes
- Sometimes we can also hide the implementation through private inner classes (in Java) or using classes that are not exported in JavaScript

Cohesion

- The degree of relatedness of an encapsulated unit
  - A measure of how much an item (a class or method) makes sense in the context in which it is placed
- Good to design classes and methods that are highly cohesive
- A class is ***highly cohesive*** if it represents one type of object and one type only
- A method is ***highly cohesive*** if it does one and one thing only
  - Method names often indicate cohesiveness
    - Strong verb/noun combination used for method names indicate high cohesiveness

Cohesion

- Example:
  - University Employees
    - Professor
    - Staff
    - Secretary
    - Registrar
  - If there are ever any changes that need to be made with respect to staff, we can go directly to the Staff class and make them there
  - We do not need to worry about affecting the code for professors
  - We do not even need to know anything about professors at all

Separation of Concerns

- Dividing a system or computer program into concerns (features and functions) that have as little overlap as possible
- Achieved through modularity (information hiding and high cohesion)
- The general rule
  - Perform everything related to a specific behavior within a single function or feature or class
    - For example, do not require client classes to call the sequence of operations needed to perform and complete a task (create a method that does this)
    - I.e., as much as possible, do everything automatically within the constructors and methods of associated classes

Single Responsibility Principle

- Separation of concerns
  - Minimal overlap with other tasks and types
- High cohesion
  - Represents one and only one thing (task or type)

- Note that when an OO system has high cohesion, its concerns (data and functionality) will have been divided into non-overlapping objects and methods, i.e., the result is separation of concerns

All Programming Languages Support Separation of Concerns

- Imperative languages like C, Pascal, Fortran
  - Concerns are separated into functions/procedures/subroutines
- Object Oriented languages like Java, C++, C#
  - Concerns are separated into objects (classes) and methods
- Business systems often separate concerns into layers
  - presentation layer
  - business logic layer
  - data access layer
  - database layer

Open System Interconnection (OSI) Reference Model

- OSI reference model is an abstract model of the design of a layered set of network protocols
- Each layer is a collection of related functions
  - receive services from the layer below
  - provide services to the layer above

- Seven Layered OSI model
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

Another Network Example

- TCP/IP layers
  - Link
  - Internet
  - Transport
  - Application

Forms of Reuse

- Collaboration
  - Simplest form of reuse (e.g., referencing an object or method of another object)
  - Every time we create or use an object of a class we are reusing the code provided by the class (written once and used by many objects)
- Inheritance
  - Should follow the Open-Closed Principle
    - Open for extension (inheritance) but closed for modification
    - Reuse methods of the superclass without modifying the structure of the algorithm
      - Example is template classes where we can extend by replacing default behavior with overriding "hook" methods but algorithm structure is unchanged
- Composition and Delegation

Reuse

- Creating classes with high cohesion
  - Results in more general objects
  - Which in turn increases reusability without changing method behavior or by extending method behavior

Reuse

- A class (once written and tested) should represent a useful unit of code which can be reused as is
  - This reusability is not nearly as easy to achieve as one would hope
  - It takes experience and insight to produce a reusable object design
  - Concerns have to be separated into a set of highly cohesive, loosely coupled classes and methods
  - Once you have such a design, reuse is easy and greatly increases productivity
- Code reuse is one of the greatest advantages of OOP

Main Point 2

A well-designed application is composed of a set of highly cohesive, self-contained modules (classes and packages). These modules are loosely coupled, i.e., they provide only an interface and hide all implementation details.

Science of Consciousness: Natural systems build complex structures from combinations of simpler self-contained structures. Contact of the mind with the deepest, most abstract level of nature brings out the qualities of coherence, order, balance, etc. into individual and collective life as demonstrated by hundreds of scientific studies.

Inheritance and Polymorphism

- E.g. a John Smith object may be a Student, a Registrar, or even a Professor
  - He is also a Person
  - Should it matter to other objects which type of Person John is?
  - It would significantly reduce the development effort if
    - other objects could treat all Person objects in the same way
    - and not be required to have a separate section of code for each type
  - The concept of polymorphism says:
    - We can treat instances of related classes uniformly (the same) throughout the system
  - The implication is:
    - we can send a message to an object without first knowing what type it is
    - the object will do “the right thing”, at least from its point of view

Dependency Inversion Principle

- Abstractions (what) should not depend on details (how)
  - Details should depend on abstractions (specifications)
- High-level modules should not depend on low-level modules
  - High-level means more abstract, more general
    - E.g., interface, abstract class, superclass
  - Low-level means more concrete, more specific/specialized
    - Concrete subclass/subtype
  - Both high-level and low-level should depend on common abstractions not each other (substitution principle)

Dependency Inversion Principle

- That is, both high-level objects and low-level objects should depend on the same abstraction
  - Thus both high- and low-level classes have a dependency on the same interface and specification, but not on each other (example: Collection, List, ArrayList, LinkedList)
- Boils down to a related Principle:
  - Program to the interface not to the implementation

Fundamental OO Concepts

- These concepts underpin good design regardless of the technology or paradigm
- Just because we have used some of these concepts before, does not mean we were doing OO
  - It just means you were doing good design
- Good design is a big part of OO, but there is a lot more to it
  - OO concepts appear deceptively simple, but ....

  Don’t be fooled
  - The underlying concepts of structured techniques also seemed simple, yet structured development was actually quite challenging
  - Just as it takes time to get good at structured development, it will also take time to get good at OO development

Problems with inheritance

- Errors in the superclass ripple down to subclasses (because they are tightly coupled)
- Superclass methods may execute different code than was originally written in the superclass (downcalls)
  - Thus supercalls may not behave as expected or may no longer terminate
- Thus subclasses are not self-contained
  - they cannot, **in general**, be understood without reference to the superclass code (non-modular because of tight coupling)

Subclassing Problems

- Downcalls can cause superclass methods
  - to behave in unexpected ways
  - to fail to terminate
- Downcalls can also cause super-calls to have the same problems
- Object aliasing can cause related problems

Downcall Example

```
class A {
   m1() { …. this.m2(); …}
   m2() { …}
}

class B extends A {
   m2() { …. }
}
```

The Java Modeling Language (JML)

- JML is a behavioral interface specification language (BISL)
  - Specifies the interface and behavior of methods
- Uses Java expressions in assertions
- Can be used to specify
  - pre- and post-conditions
  - allowed side-effects in a method
  - invariant properties of classes

Why do we need to specify the behavior of classes?

Working with Class Frameworks

- Frameworks are created so they can be extended
- Suppose we were provided with a class framework (or class library) of compiled (byte) code and specifications, but no source code
- The questions is:
  - When creating subclasses, can we reason about superclasses in the framework just from the specifications?
  - I.e., do we need the superclass code to ensure the correctness of our new subclasses?
- We're going to do a little experiment to find out

Is the subclass **CellPlusTotal** correct?

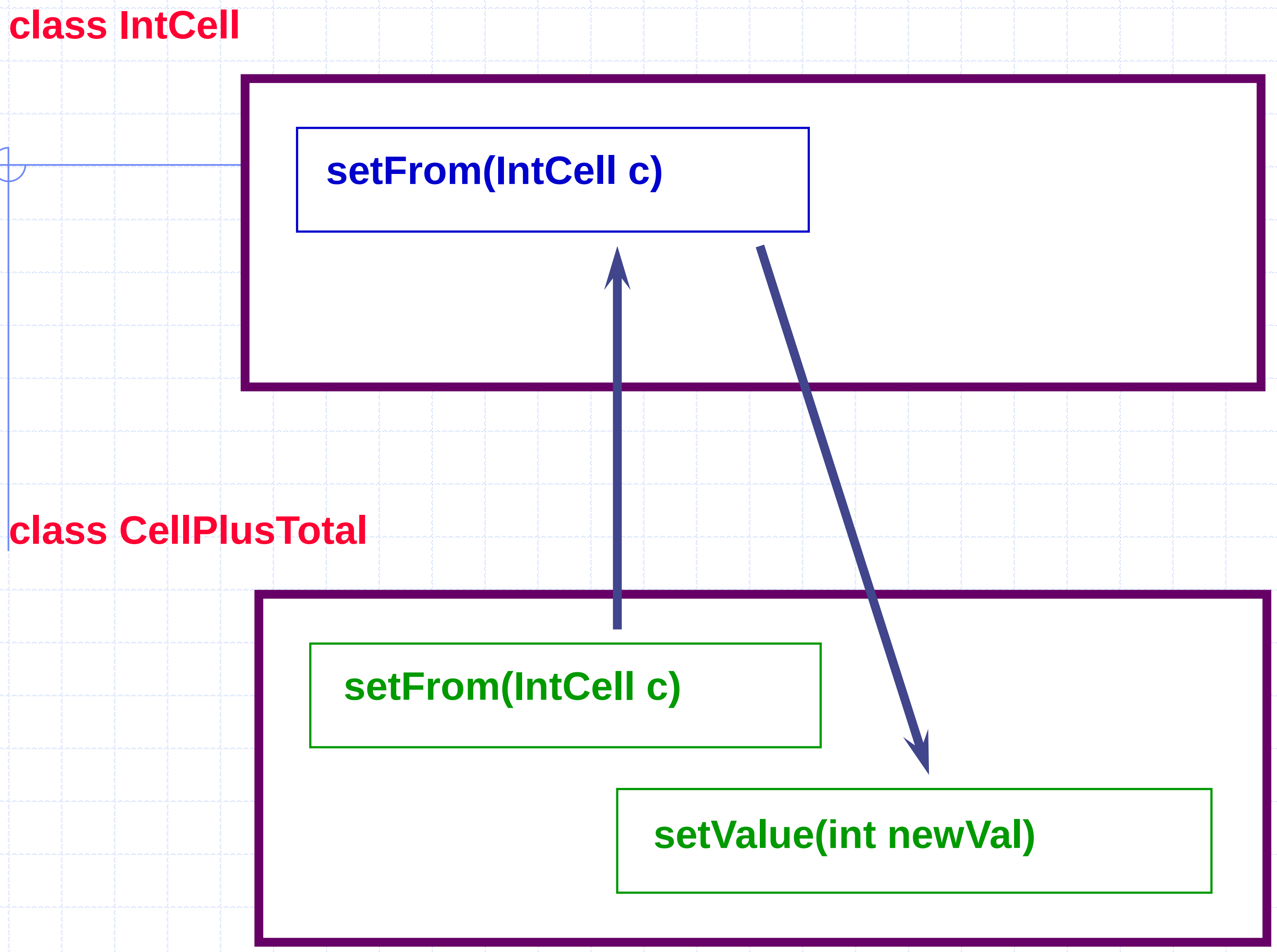
```
class IntCell {
    constructor(val) {
        this.value = val;
        this.prevVal = null;
    }
   setValue(newVal)
   { …
   }
   setFrom(IntCell c)
   {
     …
   }
// … other methods omitted
}
```

```
class CellPlusTotal extends IntCell  {
  constructor(val) {
      super(val);
      this.total = 0;
  }
  // …
  setValue(newVal) {
     super.setValue(newVal); //???
     this.total += 1;
  }
  setFrom(cell) {
     res = super.setFrom(cell);  //???
     this.total += 1;
     return res;
  }
}
```

What’s wrong with subclass **CellPlusTotal**?

```
class IntCell {
     constructor(val) {
          this.value = val;
          this.prevVal = null;
     }
 // ...
    setValue(int newVal) {
       prevVal = value;
       value = newVal;
    }
    setFrom(IntCell c) {
       setValue(c.getValue());
       return prevVal;
    }
// … other methods omitted
}
```

```
class CellPlusTotal extends IntCell  {
  constructor(val) {
       super(val);
       this.total = 0;
  }
  // ...
  setValue(newVal) {
     super.setValue(newVal); //???
     total += 1;
  }
  setFrom(cell) {
     res = super.setFrom(cell);  //???
     total += 1;
     return res;
  }
}
```


class IntCell
setFrom(IntCell c)
class CellPlusTotal
setFrom(IntCell c)
setValue(int newVal)

Problem

- Cannot reason soundly about superclass methods that may make downcalls to methods that modify subclass instance variables
  - We coined the term “*additional side-effects*” which means:
    - side-effects to subclass fields in addition to the allowed side-effects to superclass fields
- Also cannot reason soundly about overridden superclass methods that are recursive or mutually recursive (since may no longer terminate)

Simplest Way to Avoid Downcall Problems

- Any methods called as helpers by public methods, should be made private (prefix with a "_" so it is known not to override such helper methods)
- If a method is public, it can be overridden but it's behavior must be consistent with the superclass method that is being overridden (subtyping) and must not call other public methods of the class (since they can be overridden causing downcalls)
- Favor composition over inheritance whenever it makes sense
  - Do not use inheritance if there is not a "isA" relationship between superclass and subclass

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. OO applications are built by creating self-contained objects.
2. Good engineering requires that the components of a system be carefully constructed and managed to guarantee their integrity (encapsulation, high cohesion, loose coupling, separation of concerns).

3. **Transcendental Consciousness** is the silent source of all creation and remains hidden without proper techniques.
4. **Impulses within Transcendental Consciousness**: This field encapsulates the dynamic natural laws that govern creation.
5. **Wholeness moving within itself**: In Unity Consciousness, one experiences that all objects in creation arise from consciousness and are ultimately nothing but consciousness, my own consciousness.