Lecture 16

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, and maintainable. *Science of Consciousness:* Creation of any object in creation is ultimately the result of the dynamic natural laws (principles) 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 scientific research.

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

Other Related Problems

- Fragile base class problem
  - Two categories: Syntactic and semantic
- In general, superclasses cannot be changed without possibly breaking existing subclasses
  - Caused by the potentially tight coupling between subclasses and superclasses

Fragile Base Class Problem

- Syntactic FBC Problem:
  - When a superclass is changed, its subclasses and any client code may have to be recompiled
  - Caused by binary incompatibility of compiled subclasses with new releases of (newly compiled) superclasses
- Semantic FBC Problem:
  - When a superclass (or subclass) changes its behavior unexpectedly, the behavior of related components may become unpredictable

Avoiding the Semantic FBC Problem

- What developers can do:

(1) Avoid unexpected changes to the behavior of superclass methods (create subtypes)

(2) Observe rules for disciplined inheritance

(1) Avoiding unexpected changes in method behavior

Substitution Principle:

- The substitution principle requires that subclasses “behave like” the superclass
  - Clients can reason from superclass specifications without knowing the run-time type of an object

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() { … }
}

A myA = new B();
myA.m1();  // calls m2() of B
```

Downcalls in the Template Method

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

        return r[1];

…
```

```
public class Height extends EulerTour {

    visitExternal(T, p, r) {
        r[1] = 0;
    }

    visitPostOrder(T, p, r) {
        r[1] = 1 + Math.max(r[0], r[2]);
    }

    height(T) {

        return this.eulerTour(T, T.root());
    }

    …

}
```

Subclassing vs. Subtyping

- *Subclassing*: Creating subclasses
- *Subtyping*: Creating subclasses that satisfy the substitution principle
  - The subtype must have all the methods of the supertype
    - Java type system ensures this (JavaScript does not ensure this but the program will crash)
  - Subtype methods must “behave like” supertype methods
    - Not handled by Java or JavaScript, so must be done by the programmer
    - Requires a contract that specifies what the expected behavior is (and specification inheritance)
    - Subtype methods must not have side-effects if the superclass method it overrides does not
  - Subtype methods and constructors must not invalidate supertype invariant properties

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

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 are:
  - 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

(2) Rules for Disciplined Inheritance

(2) Rules for Disciplined Inheritance

- Specialization interface (Lamping)
  - The special interface between a class and its subclasses
- Typing the specialization interface (Kiczales and Lamping)
  - Defines the allowed modifications in a subclass
  - Methods are grouped and layered
    - All methods of a group have to be overridden or none can be (in the example m1 and m2 would be in the same group and both would have to be overridden)
    - Only methods in the same group or in a higher layer can be called (**no downcalls**)

(2) More Approaches to Disciplined Inheritance

- Modularity in the Presence of Subclassing
  - Raymie Stata's Doctoral Dissertation (MIT)
- Behavioral specification of the specialization interface (Stata and Gutag)
  - Methods are partitioned into disjoint groups with strict barriers
    - grouped according to the sets of fields they access and maintain
  - Method groups have to be overridden as a whole
  - Preserves modular reasoning

Another Solution

- Mikhajlov and Sekerinski (1998)
  - Defined a very restrictive set of rules that prevents the FBC problem
    - No new cyclic method dependencies between superclass and subclass methods
    - Superclass instance variables must be private
    - Subclasses cannot introduce new fields

An Advantage of Composition over Inheritance

- These downcall problems go away
  - Clemens Szyperski, et. al, Component Software: Beyond Object-Oriented Programming, 2nd Ed.
- Favor Composition & Delegation Over Inheritance (eliminates the downcall problem)
  - Never use implementation inheritance
    - Example: Vector and Stack in Java library should be composition and delegation instead of inheritance

Loose Coupling

- Loose coupling of objects
  - Methods of one object should not directly access the fields of another object
    - if the two objects are the same type, then we sometimes allow some limited direct access (i.e., as a last resort, when there is no other way),
      - but it should be avoided because this creates tight coupling between the method and the implementation of the class
- Loose coupling between superclass and subclass
  - A public method that has side-effects should not be called in the body of any other method of the same class (i.e., helper methods should be private so they are not overridden in a subclass)
    - If you want a simple rule so you don't have to think about side-effects, then one public instance method should never call another public instance method of the same class
  - An overriding subclass method must not have side-effects if the overridden superclass method has no side-effects
    - This rule must always be followed since otherwise the subclass is not a behavioral subtype of the superclass

Encapsulation

- Encapsulation is achieved through information hiding and results in loose coupling
  - Information hiding prevents access except within methods of the same class
  - In JavaScript, we precede the names of instance variables with a "_" indicating it is private

High Cohesion

- Classes must only contain information regarding one and only one conceptual entity
- Methods must perform one and only one task
  - One measure is whether the method can be named with a specific verb/noun name

Separation of Concerns

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

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 (which is non-modular because of tight coupling)

Is the subclass **CellPlusTotal** correct?

```
class IntCell {
     constructor(val) {
              this._value = val;
              this._prevVal = null;
      }
    setValue(newVal)
    {  …
    }
    setFrom(cell)
    {
      …
    }
// … 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;
  }
}
```

Our Approach (Ruby and Leavens, 2000)

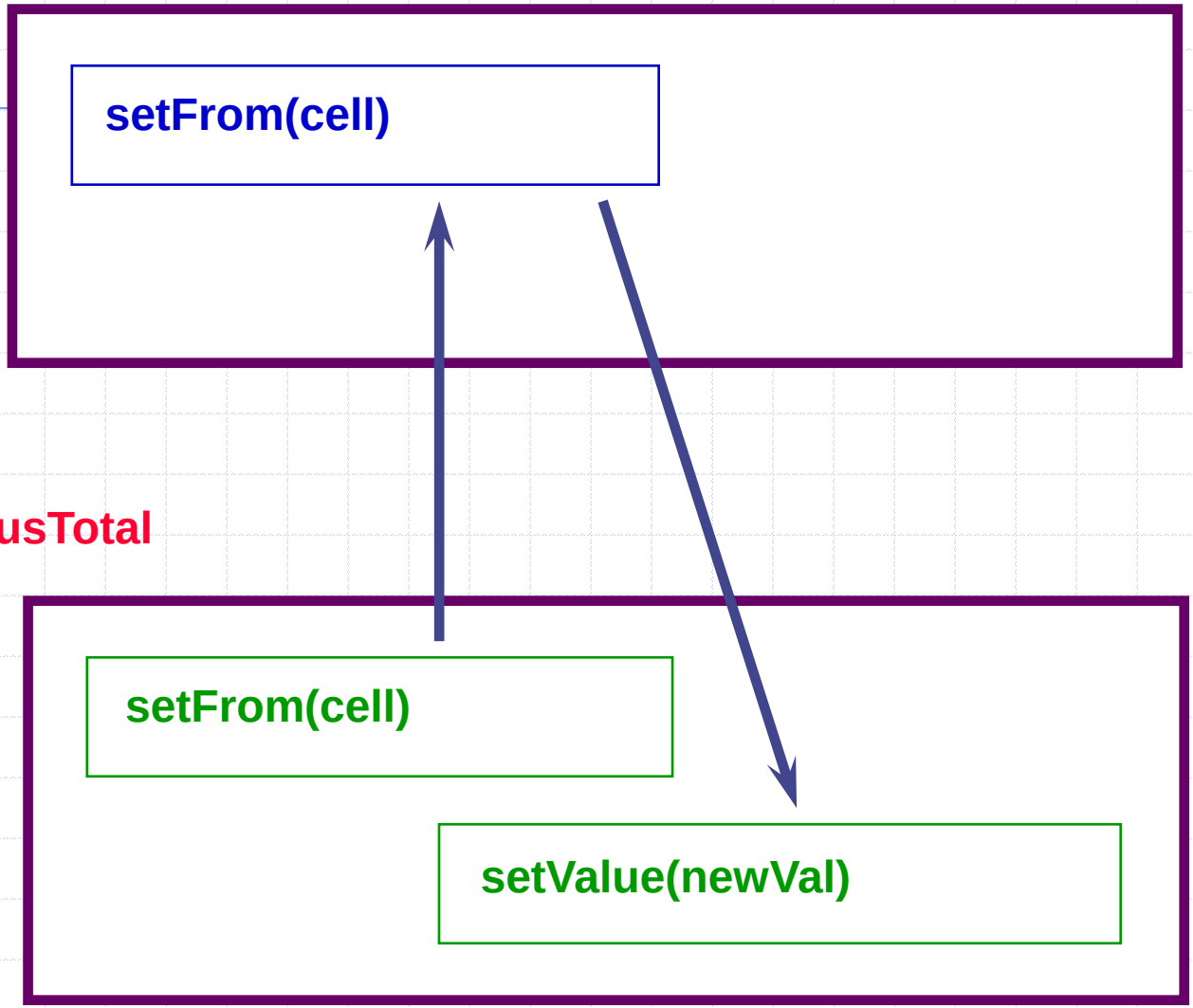
- Solves the inverse of the FBC problem
  - FBC problem says that existing unknown subclasses should not break when the superclass (base class) changes
  - Our approach assumes an existing superclass is allowed to change and that its code is unavailable to subclasses (superclass is a blackbox)
  - We defined a set of rules that allow one to reason about the correctness of a subclass without the superclass code (modular subclass verification)
- In our technique, method behavior is specified using the Java Modeling Language (JML)
  - Creates a contract between a class and its subclasses and clients
- Specification inheritance
  - prevents unexpected changes to the behavior of methods
  - ensures satisfaction of the substitution principle

Example of Downcall Problem

```
class IntCell {
    constructor(val) {
        this._value = val;
        this._prevVal = null;
    }
// ...
    setValue(newVal) {
        this._prevVal = value;
        this._value = newVal;
    }
    getValue() {
        return this._value;
    }
    setFrom(cell) { // cell is an IntCell
        setValue(cell.getValue());
        return this._prevVal;
    }
}
```

```
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;
  }
}
```

class IntCell



Possible Solutions

- Solution1: (Ruby Dissertation)
  - Provide a **code contract** that specifies the calling structure of methods in a class
    - Can be automatically generated
  - Define rules (a type system) for determining when a superclass method is unsafe to call
    - Superclass methods that may make downcalls to methods that modify subclass variables are unsafe in general
    - Prevent these calls by requiring overrides and disallowing super-calls
- Solution2: (Ruby Dissertation)
  - Never make a call to an overridable instance method that has side-effects (i.e., helper methods that change the state of the object should be private so they are not overridden)
  - A subclass method must not have side-effects if the superclass method it overrides has no side-effects (required for subtyping and substitution)
- Solution3: (Clemens Szyperski, et. al)
  - Use composition and delegation instead of inheritance to eliminate tight coupling between subclass and superclass methods

Solution 1: add **callable clause** so we can anticipate downcalls

```
class IntCell {
     constructor(val) {
              this._value = val;
              this._prevVal = null;
      }
    setValue(newVal)
    {  …
    }
//@ callable setValue
    setFrom(cell)
    {
      …
    }
// … other methods omitted
}
```

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

Solution 2: public methods only call private methods for side-effects

```
class IntCell {
    constructor(val) {
        this._value = val;
        this._prevVal = null;
    }
    _setValue(newVal) {
        this._prevVal = this._value;
        this._value = newVal;
    }
    setValue(newVal) {
        this._setValue(newVal);
    }
    setFrom(cell) {
        this._setValue(cell.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) {
      let res = super.setFrom(cell);  //???
      total += 1;
      return res;
  }
}
```

Solution 3: Composition and Delegation

```
class IntCell {
    constructor(val) {
        this._value = val;
        this._prevVal = null;
    }
// ...
    setValue(newVal) {
        this._prevVal = value;
        this._value = newVal;
    }
    getValue() {
        return this._value;
    }
    setFrom(cell) { // cell is an IntCell
        setValue(cell.getValue());
        return this._prevVal;
    }
}
```

```
class CellPlusTotal {
  constructor(val) {
      this._myCell = new IntCell(val);
      this._total = 0;
  }
  // ...
  setValue(newVal) {
      this._myCell.setValue(newVal);
      this._total += 1;
  }
  setFrom(cell) {
      let res = this._myCell.setFrom(cell);
      this._total += 1;
      return res;
  }
}
```

An Advantage of Composition over Inheritance

- These downcall problems go away
  - Clemens Szyperski, et. al, Component Software: Beyond Object-Oriented Programming, 2nd Ed.
- Favor Composition & Delegation Over Inheritance (downcall problems go away)
  - Never use implementation inheritance
    - Example: Vector and Stack in Java library should be composition and delegation instead of inheritance

Important Principles in OO Design (Review)

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

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

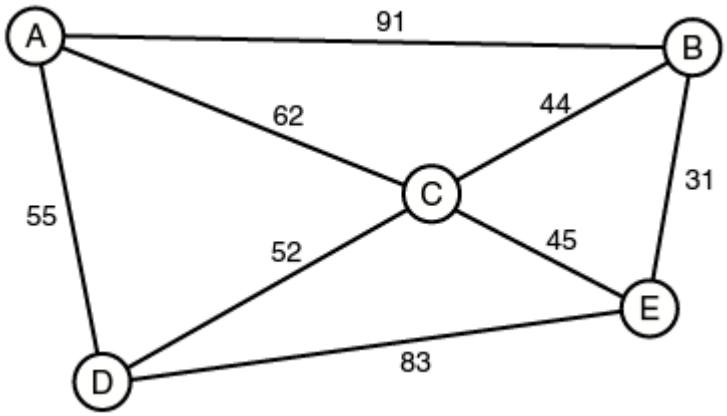
  - The underlying concepts of structured techniques also seemed simple, yet structured development was actually quite challenging
  - Just as it takes time to get good at structured development, it will also take time to get good at OO development

Intractable Problems

- In this course, we have seen Big Oh values ranging from O(1), O(n), O(nlog n), $O(n^2)$, up to $O(n^3)$.
- Algorithms with these big Oh values can be used to find solutions to most practical problems.
- However, some algorithms have big Oh values that are so large that they can be used only for relatively small values of N. For example $O(2^n)$, O(n!), etc. are impractical. Problems that have these running times are said to be ***intractable***.
- We will see two Intractable Problems next.

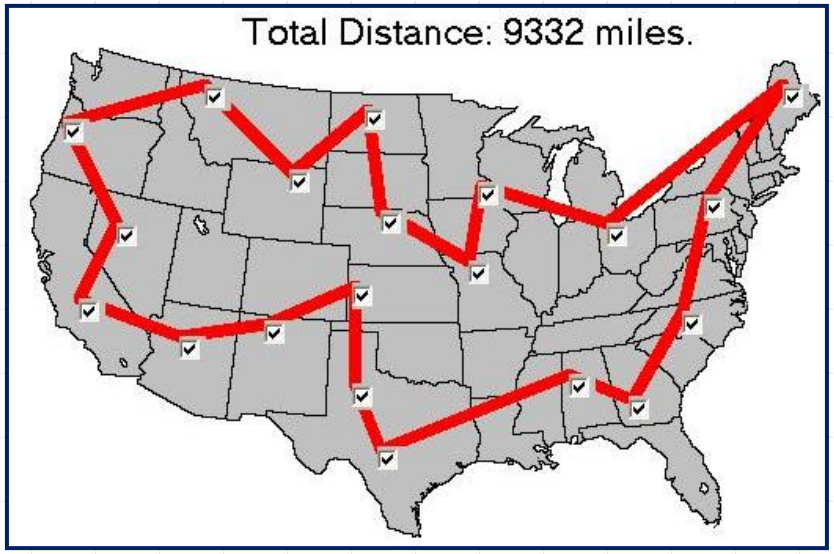
Hamiltonian Cycles

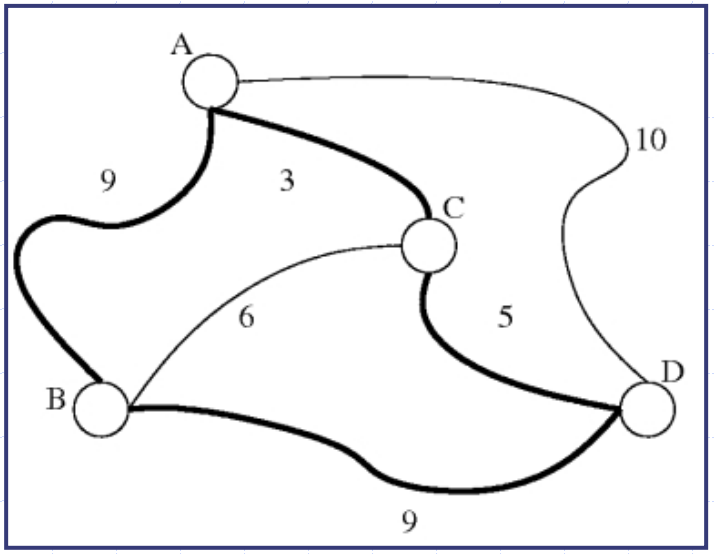
- A Hamiltonian cycle in a graph G is a simple cycle that contains every vertex of G. A graph is a Hamiltonian graph if it contains a Hamiltonian cycle.
- Exercise:



Another famous Example: Traveling Salesman Problem

- *Traveling Salesman Problem* (TSP): Given a complete graph G with cost function c: E $\rightarrow$ N and a positive integer k, is there a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is at most k? Note that a solution is always a subset of E.





Handling Intractable Problems

- Much of the research in the field of algorithms is dedicated to finding ways to handle intractable problems
- Some approaches that are currently used include
  - Approximation algorithms – devise faster algorithms to solve these problems with outputs that are not optimal but in some cases may be good enough
  - Probabilistic algorithms. Probabilistic algorithms can often execute very efficiently and output results that are correct with high probability.
  - Using intractable problems as an advantage. Modern day cryptosystems use intractable problems to prevent hackers from accessing protected resources.

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.