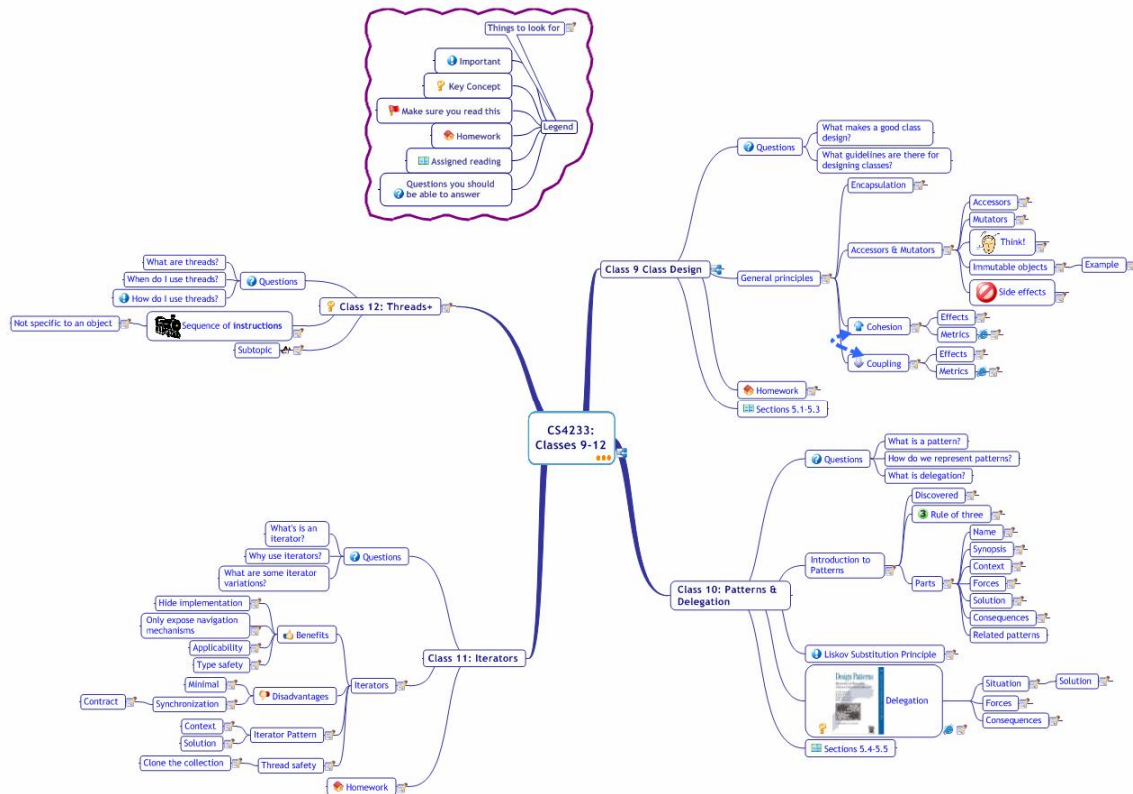


# CS4233: Classes 9-12



See document: [CS4233 Object-Oriented Analysis and Design.mmap](#)

## 1 Class 9 Class Design

See document: [CS4233 Object-Oriented Analysis and Design.mmap](#)

### 1.1 Questions

#### 1.1.1 What makes a good class design?

#### 1.1.2 What guidelines are there for designing classes?

### 1.2 General principles

There are some general principles that can be stated about designing classes. Several of these are described in this section.

However, when we talk about general principles, it's important to remember that any principle or best practice only applies in the right context. There are no universally applicable guidelines.

#### 1.2.1 Encapsulation

While encapsulation is a fundamental attribute of the Object-Oriented paradigm, it also describes a fundamental principle of good class design; namely, *hide all implementation details from the user of the class*.

The reason for doing this is so that you can change the underlying implementation without requiring user changes.

A class that makes internal representations visible is usually poorly designed.

## 1.2.2 Accessors & Mutators

Also called *getters and setters*.

The usual way of accessing the properties (attributes) of a class. Good encapsulation will hide the data representation. The user of a class should be unaware of whether there is an actual field in the object for a property or if the property is calculated. The accessors and mutators are the interfaces to the properties.

### Accessors

Accessors retrieve the property. An accessor should not have any side effects. This means that an accessor should not change the state of the property's object.

Further, it is not good practice to return a property as a value that, if you change it, will be reflected in the original object. For example, assume object A has a Vector, v, that it uses to store some set of items and provides an accessor method, getV(). Now, if getV() returns the reference to the actual vector v, the caller to getV() can modify the contents of the vector.

Unless there is a critical need to allow such modifications, you should return a clone of the vector.

### Mutators

Mutators (or setters) are methods that allow (controlled) modification of properties. In effect, the mutators change the state of the object.

Mutators should also be very specific in their effect. They should *only* modify the property specified and cause no other side effects to the state of the object.

### Think!

Should you provide accessors and mutators for every property? There are several disadvantages to doing so.

First of all, you may not need them. Whenever you provide an accessor or mutator to a property, you are telling other programmers that they are free to use them. You have to maintain these methods from that point on.

Second, you may not want a property to change. If so, don't provide a mutator.

### Immutable objects

There are times when you want an object to be created in a certain state and not allow anyone to change it. These are called *immutable* objects.

An immutable object will usually have its state set at construction time and provide only accessor methods to its users.

### Example

In the RemoteMower application, the RMStatus class is immutable.

### Side effects

Side effects are signs of poor design. If a method is supposed to cause a side effect, it should be a separate method (not a mutator per se), and should be named appropriately.

### 1.2.3 Cohesion

See also: [Coupling](#)

Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together". As such, cohesion is a semantic concept.

One way of determining whether a class is designed well is to ask if it is cohesive. If you find that the class can be separated into different classes, each more cohesive than the original, then the class is not designed well.

You want your classes to have a *high* degree of coupling.

#### Effects

Class cohesion can dramatically effect the design, understandability, and maintainability of a software system. A class that is cohesive will usually exhibit a locality characteristic. That is, a change to certain behavior in the system will usually be isolated to a single, or small number of places.

#### Metrics

See document: [index-2.html](#)

The primary metric to measure cohesion is the Lack of Cohesion of Methods (LCOM) that was first described by Chidamber and Kemerer. It has since been refined into different computations. The link associated with this topic contains a good description of the different LCOM metric computations.

### 1.2.4 Coupling

See also: [Cohesion](#)

Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

When two classes are coupled there is at least a one-way dependency between them. This means that a change in one may require a change in the other. Consider what this would mean when you have a chain of classes that are coupled.

You want your classes to have a *low* degree of coupling.

#### Effects

As with low cohesion, high coupling can cause problems with understandability and maintainability. Changes in one class can cause a chain reaction of changes. Tests will break and you will have problems isolating the errors.

#### Metrics

See document: [index-2.html](#)

The primary metric for coupling is the Coupling Between Objects (CBO) metric that was first presented by Chidamber and Kemerer. CBO is described on the same page as the LCOM metric description.

## 1.3 Homework

Do Exercise 3.10 in the text. With this class, provide a set of JUnit tests that exercise all of your code.

Turn in the complete Eclipse project zipped. Your class should be `wpi.edu.cs.cs4233.time.TimeOfDay`. Your test class(es) should be in the same package.

## **1.4 Sections 5.1-5.3**

# **2 Class 10: Patterns & Delegation**

## **2.1 Questions**

### **2.1.1 What is a pattern?**

### **2.1.2 How do we represent patterns?**

### **2.1.3 What is delegation?**

## **2.2 Introduction to Patterns**

You should be somewhat familiar with the general concept of design patterns from software engineering.

Software design patterns are "patterned" after the architectural patterns described by Christopher Alexander et al., in *A Pattern Language: Towns, Buildings, Construction*.

The principles were first applied to software design by Erich Gamma in his doctoral dissertation, and then refined further in the book *Design Patterns*, by Gamma, Johnson, Helms, and Vlissides.

### **2.2.1 Discovered**

Patterns are not invented. They are discovered. When you examine proven solutions to common problems, you can describe them by a pattern.

### **2.2.2 Rule of three**

Once is an event. Twice is a coincidence. Three times is a (potential) pattern.

### **2.2.3 Parts**

There are several parts to a pattern description. While there is not a universal set of parts to a pattern, some common ones are described in the following paragraphs.

#### **Name**

Every pattern must have a name that is descriptive as to its purpose.

#### **Synopsis**

This section is a one or two-sentence description of the pattern. It conveys the essence of the solution provided by the pattern.

The synopsis is directed to experienced programmers.

#### **Context**

This section describes the problem that the pattern addresses.

## Forces

This section summarizes the considerations that lead to the general solution.

## Solution

This is the core of the pattern. It describes a general-purpose solution to the problem.

## Consequences

Describes the positive and negative implications of applying the pattern.

This section may also contain variations on the solution.

## Related patterns

### 2.3 Liskov Substitution Principle

The Liskov Substitution Principle was described by Barbara Liskov at MIT. Basically, the LSP says:

*If for each object  $o1$  of type  $S$  there is an object  $o2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behaviour of  $P$  is unchanged when  $o1$  is substituted for  $o2$  then  $S$  is a subtype of  $T$ .*

There is a good paper describing the LSP at

<http://www.objectmentor.com/resources/articles/lsp.pdf>. If you want to understand all of the details, I recommend reading this paper.

### 2.4 Delegation

See document: [view.jsp](#)

Delegation is one way of extending the behavior of a class without using inheritance. It is often more flexible in one sense (you don't have to know much about the parent class) and more restrictive in another (polymorphism doesn't work).

Although it is not one of the primary design patterns, it is a very important concept.

Since the inheritance relationship is defined at compile-time, a class can't change its superclass dynamically during program execution. Moreover, modifications to a superclass automatically propagate to the subclass, providing a two-edged sword for software maintenance and reuse. In summary, inheritance creates a strong, static coupling between a superclass and its subclasses.

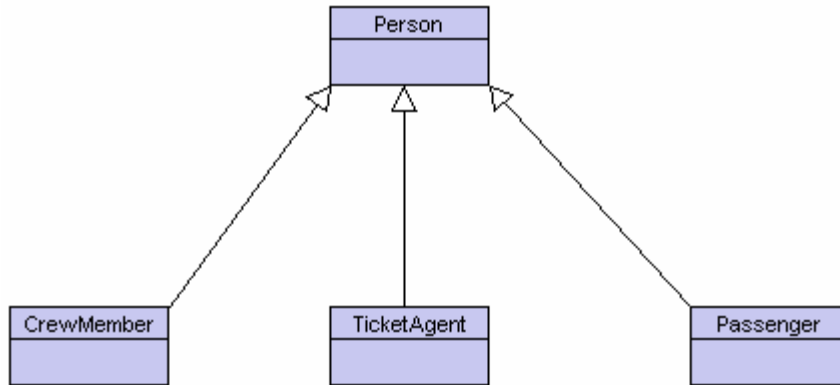
Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate. Delegation can also a powerful design/reuse technique. The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time. But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate dynamic polymorphism.

#### 2.4.1 Situation

When you want to extend a class, the usual way of doing this is through subclassing. However, this is not always possible. Inheritance is good for capturing is-a-kind-of relationships.

Sometimes we want to have an *is-a-role-played-by* relationship.

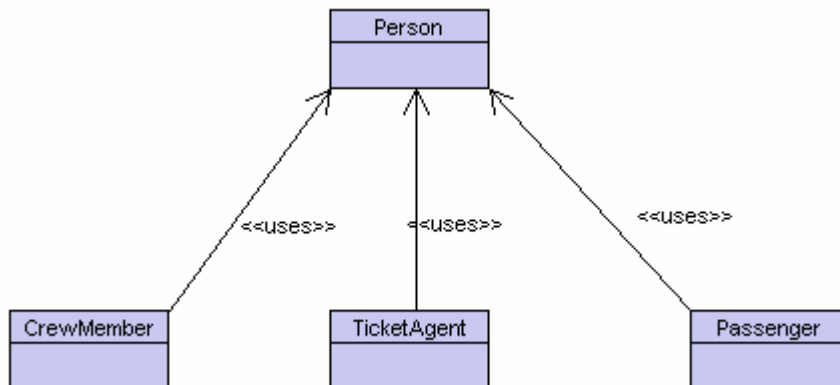
Consider the following diagram that illustrates one way of modeling people in an airline reservation system.



A problem can arise when you have a person, such as a person who is a pilot on one flight and a passenger on a different flight. If you create the object for the person as an instance of a CrewMember, what do you do when that person is a passenger on a different flight? If you create a different object that is an instance of Passenger, how do you know that it is the same person as the pilot?

## Solution

A better way to model the situation is shown in the following diagram.



This solution allows you to assign multiple roles to a person simultaneously.

The three specific classes delegate behavior to the Person class. The person is the *Delegate* and the other classes are *Delegators*.

### 2.4.2 Forces

If you find that you have a class that you think might be a subclass of another class, yet attempts to hide methods or variables from the superclass, then inheritance is not appropriate.

Sometimes you want to subclass a class that is declared as *final*.

### 2.4.3 Consequences

- Easy to compose behavior at runtime.
- Less structured than inheritance. In the previous example, a CrewMember is not a kind of a Person.
- You don't have to implement all methods (as passthroughs) of the Delegate.

## 2.5 Sections 5.4-5.5

# 3 Class 11: Iterators

## 3.1 Questions

### 3.1.1 What's is an iterator?

### 3.1.2 Why use iterators?

### 3.1.3 What are some iterator variations?

## 3.2 Iterators

An *iterator* is a commonly used mechanism for navigating over collections. As we learned more about good abstraction and encapsulation principles, iterators have become more popular. There are consequences of iterators that may not be evident, and design decisions that must be made when using them.

### 3.2.1 Benefits

#### Hide implementation

Iterators hide the implementation of both the underlying data structure, and the mechanism used to traverse that data structure.

#### Only expose navigation mechanisms

Related to hiding the implementation, iterators do not usually give the iterator client the ability to change the contents / structure of the underlying collection's navigation mechanism (such as links) in an *uncontrolled* way.

#### Applicability

Iterators can be used for many types of data structures, such as trees and graphs, that can have quite different ways of retrieving the individual elements.

#### Type safety

Iterators can be used to create type-safe collections that only allow the addition and retrieval of objects that are instances of specific classes.

### 3.2.2 Disadvantages

#### Minimal

Because iterators are usually intended for use with different underlying collections, they cannot provide all of the capabilities that the underlying collection might provide. For example, a linked list allows you to insert items in an arbitrary position. The iterator interface cannot provide this for any generic collection.

The Java Iterator interface has only three methods: `hasNext()`, `next()`, and `remove()`. Notice that `remove()` is optional. Is this good or bad?

Could there be more methods without losing generality?

## Synchronization

Synchronization between the iterator and the underlying collection can be a problem. What if some operation changes the collection while another method is using an iterator to traverse the collection?

There can be serious problems when this happens. Iterators should be safe in one of two ways:

- The iterator should be aware of the change in the underlying collection, or
- The iterator should work on a copy of the collection at the time the iterator was created.

## Contract

It is possible to use the *programming-by-contract* paradigm when implementing iterators. In this case, you can indicate in the contract's preconditions that the operation is unsafe in the presence of change and then let the user worry about this.

What do you think of this solution? What are the benefits and disadvantages?

## 3.2.3 Iterator Pattern

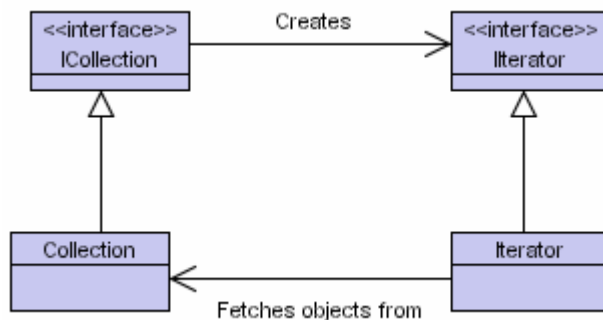
The Iterator pattern is a structural pattern that was described in the GoF (Gang of Four) book.

### Context

Suppose you are writing classes that allow someone to browse a set, like inventory in a warehouse. They can see description, quantity on hand, and so on. You want to allow the user to sequentially access the inventory items.

### Solution

The solution to the problem described in the context is the Iterator pattern that is shown in the following diagram.



## 3.2.4 Thread safety

One typical problem with collections and iterators is that the underlying collection can be modified by other threads at inopportune times. There is a `Collections` class with static methods that implements thread-safe collections.

### Clone the collection

When you want the iterator to be type safe, you need to clone the collection and use the cloned collection to iterate over.

In order to do this, you should synchronize on the collection while you create the clone. That will block other threads from modifying the collection before the clone is created. Now your iterator on

the clone is guaranteed to work properly (without, of course, any changes to that might be made to the original collection after the clone is made).

### **3.3 Homework**

Write a program that implements an iterator over a TreeMap. The iterator should return the objects stored in the TreeMap according to the order they are added to the map, regardless of Key/Value. Make your collection only accept objects of a specific type (I don't care what type, you can just subclass Object to the class TestObject that has no behavior). The iterator should return objects of type TestObject when it gets the next item in the collection.

Write JUnit tests to prove that your code works and submit the zipped Eclipse project using turnin.

turnin cs4233 hw6 homework6.zip

Due by class 16.

## **4 Class 12: Threads+**

Most students have studied thread in previous courses, but have not necessarily had to design an application that uses them. This class will address threads and, if time permits, begin to discuss the patterns scheduled for class 13.

### **4.1 Questions**

#### **4.1.1 What are threads?**

#### **4.1.2 When do I use threads?**

#### **4.1.3 How do I use threads?**

### **4.2 Sequence of instructions**

A thread is a sequence of instructions that executes independently of other instructions *within a single process*. Threads are often called "lightweight processes."

#### **4.2.1 Not specific to an object**

Threads are not specific to any single class or object. The instructions in a thread span many objects, as necessary.

Each thread has its own call stack that represents the methods that are active, and they can be from different objects.

### **4.3 Subtopic**

See document: [index.html](#)

Most of what you need to know about how to use threads in Java is in the associated link. I am not going to add text for the sake of making these notes look larger than necessary. You should read the information from the Java tutorial in that link.