CSE 223: Programming -2
04-Design Styles & Basic Design Patterns

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- 3 layers architecture
- Model-View-Controller (MVC)
- Inversion of Control (IOC) – Dependency Injection (DI)
- Introduction to Design Patterns
- Basic Design Patterns

3 layers architecture

3 layers architecture

- Architecture styles for decomposing a system? (not exhaustive)
- Three-tier architecture (in 1970s for information systems)
- **Interface** layer: boundary objects that deal with the user (windows, forms, web pages, ..)
- **Application** logic layer: control objects, realizing the processing, rule checking, and notification required by the application
- **Storage** layer: storage, retrieval, and query of persistent objects
- Allows development of different user interfaces for same application logic
- **Object** Model to capture the attributes of the business model
  - Students, Professors: not their location on the screen or their color
  - Usually as Java Beans
- Interface layer depends on Application layer and object model
- Application layer depends on Storage layer and object model
- Storage layer depends object model
- See *Spring* framework

Model-View-Controller (MVC)

Problem

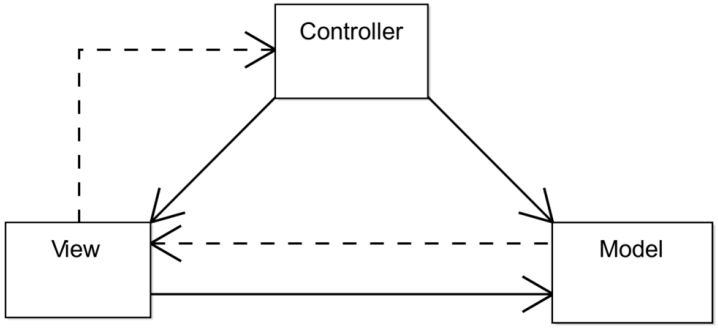
- Users interact with data
- Data is manipulated according to some business logic
- Data should be validated for consistency
- We may need different views
  - Graphical User Interface (GUI)
  - Text mode
  - Mobile view

Simple solution

- A class for holding data
  - Student
  - Course
- A class(es) for displaying data and taking user input
- GUI classes loads/stores data on startup/shutdown
- Cons
  - Business logic in GUI classes
  - A lot of code duplicated for different views
  - GUI “knows” where and how data is stored
  - Ripple effect
- Pros
  - Simple (as its name states)

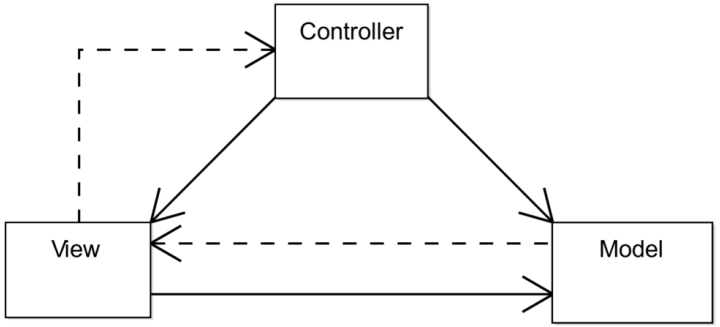
Model-View-Controller (MVC)

- Decouples presentation code from data access code
- Allows multiple views with minimum code rewrite
- Can easily change data access code without rewriting business logic code
- Model
  - Data access code
  - SQL
  - Raw files
  - Validation
  - Unique ID
  - Name length
  - Relationship
- ORM Libraries → see Hibernate



Model-View-Controller (MVC)

- View
  - Displays data in an appropriate format
  - Takes user input
  - Accesses application sent by controllers
- Controller
  - Generates data from model to be sent to views
  - Business logic implementation
  - Use user input collected from views
  - Updates data through model
- Examples
  - VueJS
  - AngularJS
  - Swing
  - Ruby on Rails
  - Java EE



Inversion of Control (IoC)
Dependency Injection (DI)

Inversion of Control (IoC)
Dependency Injection (DI)

```
public class Car
{
        private Engine engine;

        public Car(){
                this.engine = new Engine();
        }
}
```

The car makes a new engine when the car is being constructed

```
public class Car
{
        private IEngine engine;

        public Car(IEngine engine) {
                this.engine = engine;
        }
        public void Drive() {
                this.Engine.GoFaster();
        }

}
```

We move the creation of the dependency into whatever creates the object in the first place

The created engine is now slotted into the car when the car is being constructed

But what if we wanted to swap this for an ElectricEngine? What about a DieselEngine?

Benefits

- Better separation of code
- Better testability through stubs and mock objects
- The object doesn’t need to know implementation details of its dependency
- Unit Testing
  - If we want to test our Car class, we need an IEngine
  - Unit testing is supposed to be of a small unit of code
  - We can’t unit test the Drive() method…
  - … unless we have an engine which has no side effects!
  - We use a mock or a stubbed-out implementation of the engine which does nothing.
  - We simply pass this mock in with DI

Unit testing for IoC

```
[Test]
public void TestDriveCar()
{
        IEngine engine = new MockEngine();
        Car myCar = new Car(engine);
        // Check the Drive method does the
        // correct thing…
        myCar.Drive()
}
```

- Dependencies don't have to be injected through the constructor.
- It's acceptable to pass in dependencies through properties or getter/setter methods too
- Examples
  - The Spring framework

Introduction to Design Patterns

Design Pattern

- What?
  - *A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever using it the same way twice.*
  - *Reusable solutions to recurring problems that we encounter during software development.*
- Why?
  - *Designing object-oriented code is hard, and designing reusable object-oriented software is even harder.*
  - Patterns enable programmers to *recognize a problem and immediately determine the solution without having to stop and analyze the problem first.*
  - Provides a framework for communicating complexities of OO design at a high level of abstraction
  - When we understand a pattern well enough to put it into words, we are able to intelligently combine it with other patterns.
  - A pattern can be used in discussions among programmers who know the pattern.

History

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns
  - The GoF book describes a pattern using the following four attributes:
    - **name** to describes the pattern, its solutions and consequences in a word or two
    - **problem** describes when to apply the pattern
    - **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
    - **consequences** are the results and trade-offs in applying the pattern

Definition of a Design Pattern--2002

- Pattern name—same as GoF attribute
- Synopsis—conveys the essence of the solution
- Context—problem the pattern addresses
- Forces—reasons to, or not to use a solution
- Solution—general purpose solution to the problem
- Implementation—important considerations when using a solution
- Consequences—implications, good or bad, of using a solution
- Java API usage—examples from the core Java API
- Code example—self explanatory
- Related patterns—self explanatory

Classifications of Design Pattern

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

Fundamental patterns

- “The most fundamental and important design patterns to know”
  - Most other patterns use at least one of these
  - So ubiquitous that they’re often not mentioned
- List
  - Delegation—when not to use Inheritance
  - Interface
  - Abstract superclass
  - Interface and abstract class
  - Immutable
  - Marker interface
  - Proxy

Creational Patterns

- Provides guidance on how to create objects when their creation requires making decisions
  - “Decisions typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to”
  - Value is to tell us how to structure and encapsulate the decisions
- List
  - Factory method
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton
  - Object pool

Partitioning Patterns

- Follows the divide and conquer paradigm
  - “Provide guidance on how to partition classes and interfaces in ways that make it easier to arrive at a good design”
- List
  - Filter
  - Composite
  - Read-only interface

Structural Patterns

- *Describe common ways that different types of objects can be organized to work with each other*
- *Structural patterns help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.*
- List
  - Adapter
  - Iterator
  - Bridge
  - Façade
  - Flyweight
  - Dynamic linkage
  - Virtual proxy
  - Decorator
  - Cache management

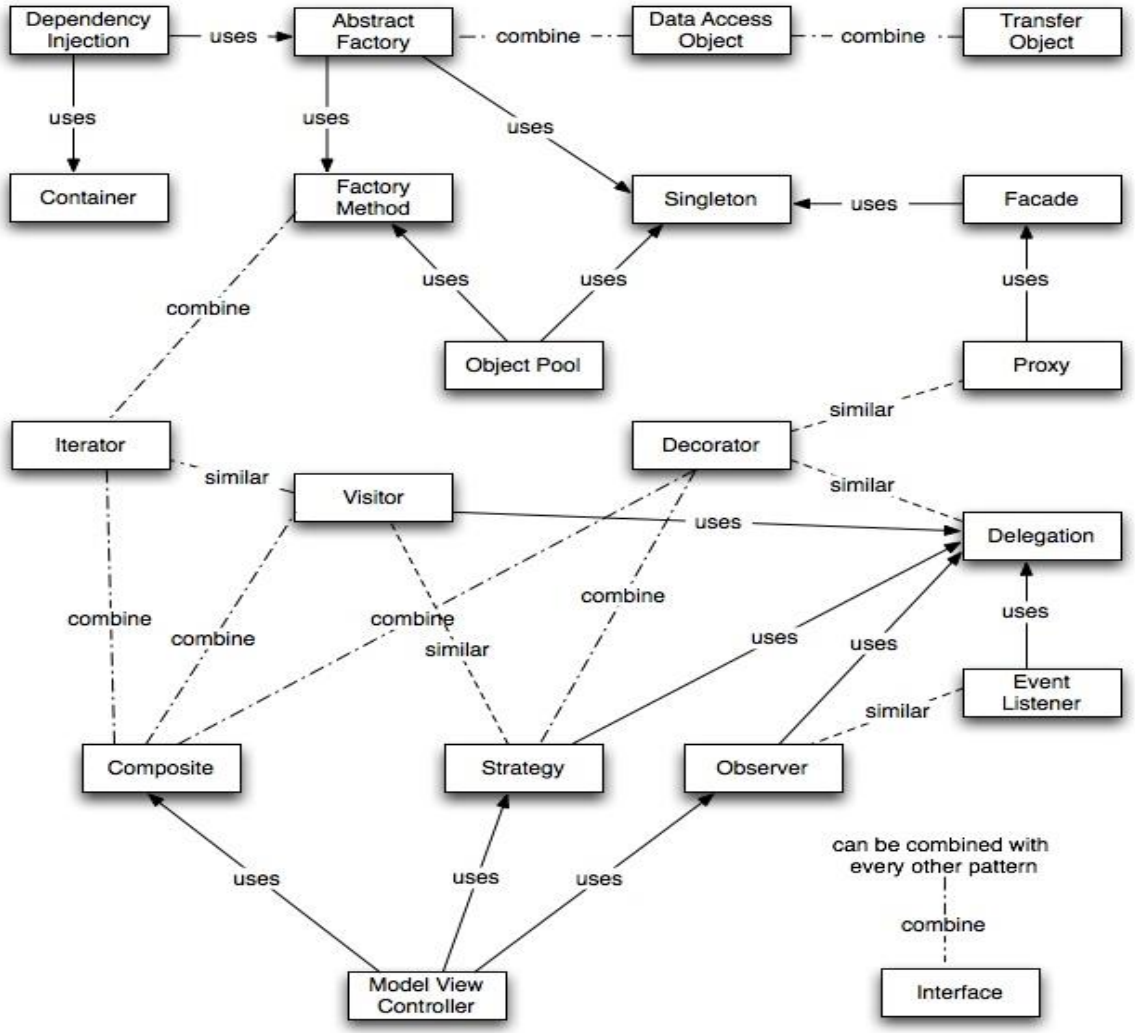
Behavioral Patterns

- *They are used to organize, manage, and combine behavior*.
- *They help you define the communication between objects in your system and how the flow is controlled in a complex program.*
- List
  - Chain of responsibility
  - Command
  - Little language
  - Mediator
  - Snapshot
  - Observer
  - State
  - Null object
  - Strategy
  - Template method
  - Visitor

Concurrency Patterns

- These patterns involve coordinating concurrent operations and address two primary problems
  - Shared resources
  - Sequence of operations
- List
  - Single threaded execution
  - Lock object
  - Guarded suspension
  - Balking
  - Scheduler
  - Read/Write lock
  - Producer-consumer
  - Two-phase termination
  - Double buffering
  - Asynchronous processing
  - Future

Design Patterns Map



Fundamental Design Patterns

Fundamental Design Patterns

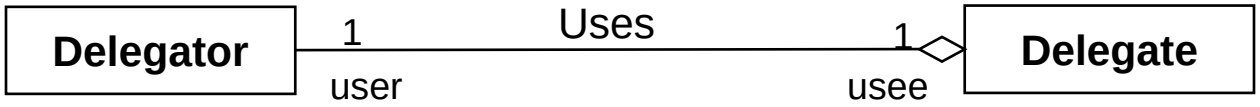
- The most fundamental and important design patterns to know. You will find these patterns used extensively in other design patterns.
- The Delegation; Interface; Abstract Superclass; and Interface and Abstract Class patterns demonstrate how to organize relationships between classes.
- The Immutable pattern describes a way to avoid bugs and delays when multiple objects access the same object.
- The Marker Interface pattern describes a way to simplify the design of classes that have a constant boolean attribute.
- The Proxy pattern is the basis for a number of patterns that share the common concept in which an object manages access to another object in a relatively transparent way.

Delegation (When Not to Use Inheritance)

- Intent
  - A way to extend and reuse functionality of a class by writing an additional class with added functionality that uses instances of the original class to provide the original functionality.
- Motivation
  - Inheritance is a common way to extend & reuse functionality.
    - Usually is-a-kind-of (Specialization)
  - Inheritance is inappropriate for many situations:
    - is-a-role-played-by ?
  - Delegation is a more general way for extending a class's behavior.
- Applicability
  - Use the Delegation pattern when
    - Behavior might change over time.
    - Not conceptually an inheritance relationship.

Delegation

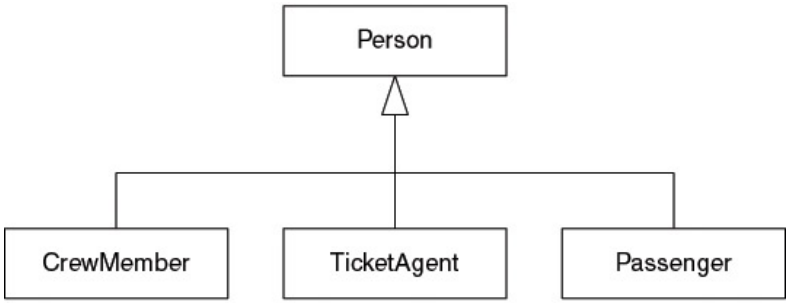
- Structure

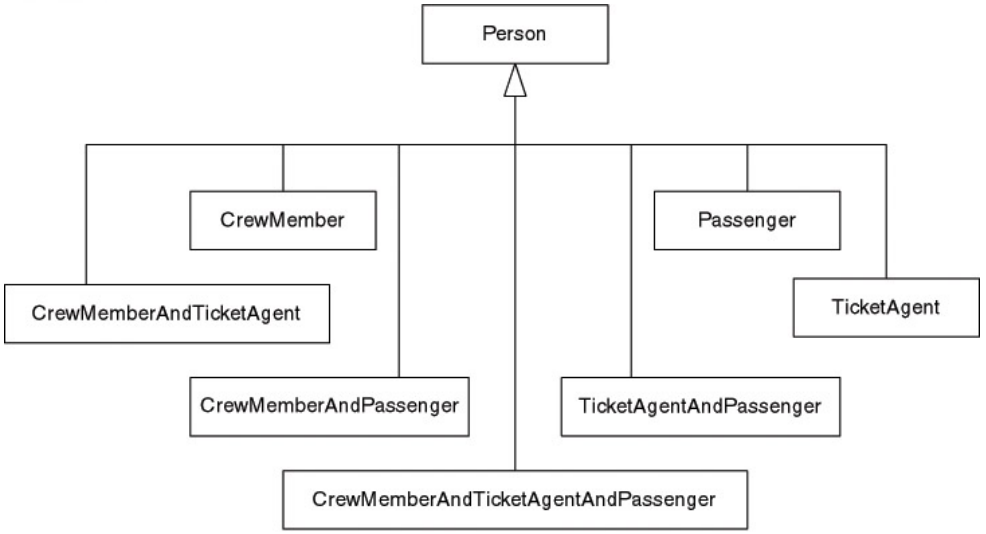


- Participants
  - Delegator
    - declares interface for clients.
    - delegates messages to `Delegate`.
  - Delegate
    - answers to messages sent by the `Delegator`.
    - may be shared.
- Collaborations
  - Clients use the Delegator interface to send messages to the Delegator, as a side effect, the same message is sent to the Delegate by the Delegator.

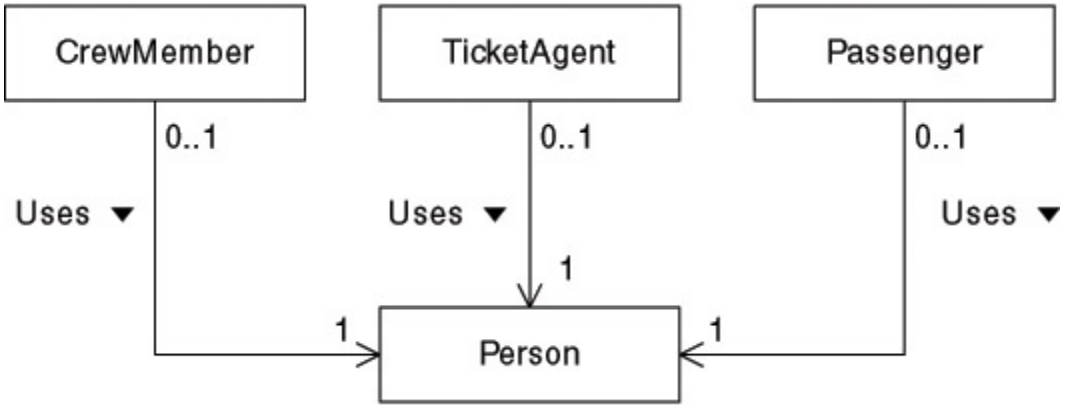
Delegation

- Consider the example of an airline reservation system that includes such roles as passenger, ticket-selling agent, and flight crew.





Delegation


CrewMember
TicketAgent
Passenger
0..1
0..1
0..1
Uses
Uses
Uses
1
1
1
Person

Interface

- Intent
  - Keep a class that uses data and services provided by instances of other classes independent of those classes by having those instances through an interface.
- Motivation
  - Capture common behavior semantics
  - Assists “Programming in the large”
    - commit to a set of interfaces and their semantics between software teams.
- Applicability: Use the Interface pattern when
    - Common behavior may have different implementations.
    - Business protection issues - not reveal actual code.
    - Improving parallelism

Interface

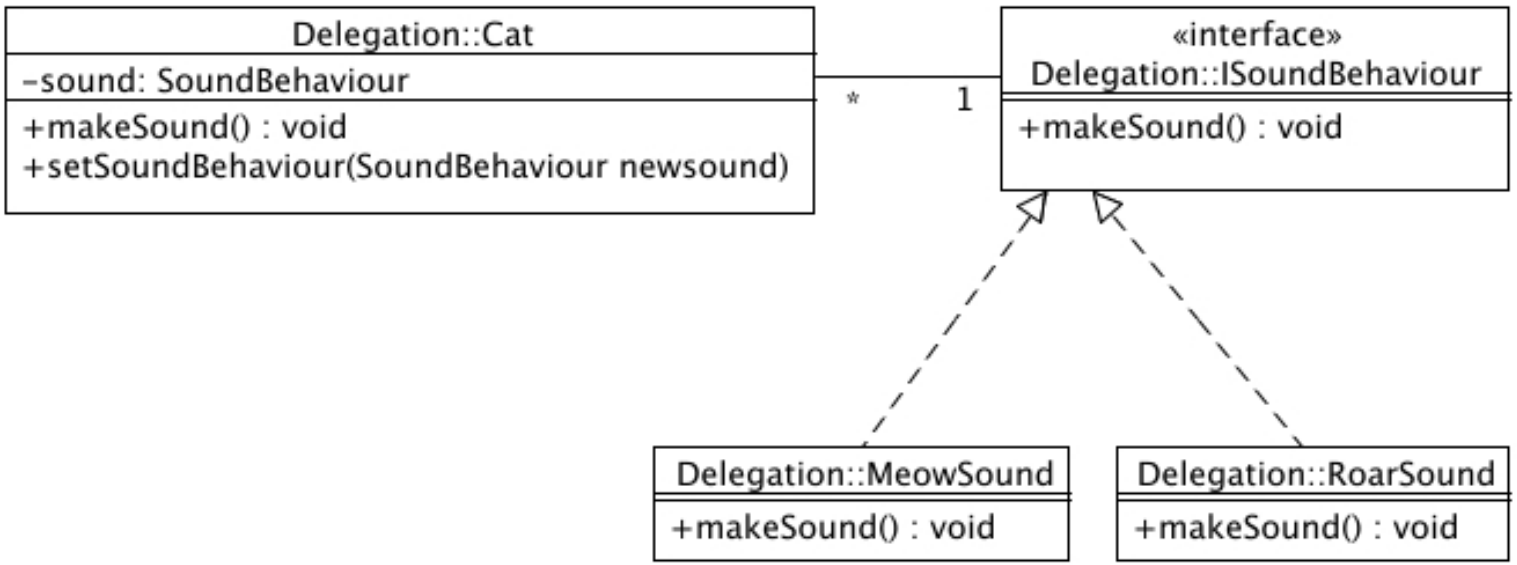
- Structure

- Participants
  - Client
    - uses `Service` classes that implement `IndirectionIF`.
  - IndirectionIF
    - provides the indirection that keeps the `Client` class independent of the `Service` class.
  - Service
    - provides a service to `Client` classes by implementing the `IndirectionIF` interface
- Collaborations
  - Clients use the interface to interact with different service objects which are instances of different classes which are guaranteed to implement the interface.

Interface

- Consequences
  - Advantages:
    - keeps a class that needs a service from another class from being coupled to any specific class.
  - Disadvantages:
    - like any other indirection, the interface pattern can make a program more difficult to understand.
    - A class's constructors cannot be accessed through an interface, because Java's interfaces cannot have constructors.
- Implementation
  - The implementation of an interface in Java is very straightforward.
  - In C++, you can write a pure abstract class and supply no method implementation to imitate Java interfaces

Example: Delegation + Interface



Code Example

```
public interface ISoundBehaviour {

	public void makeSound();
}
public class MeowSound implements
ISoundBehaviour {

	public void makeSound() {

System.out.println("Meow");
	}
}
public class RoarSound implements
ISoundBehaviour {

	public void makeSound() {
		System.out.println("Roar!");
	}
}
```

```
public interface ISoundBehaviour {

	public void makeSound();
}
public class MeowSound implements
ISoundBehaviour {

	public void makeSound() {
		System.out.println("Meow");
	}
}
public class RoarSound implements
ISoundBehaviour {

	public void makeSound() {
		System.out.println("Roar!");
	}
}
```

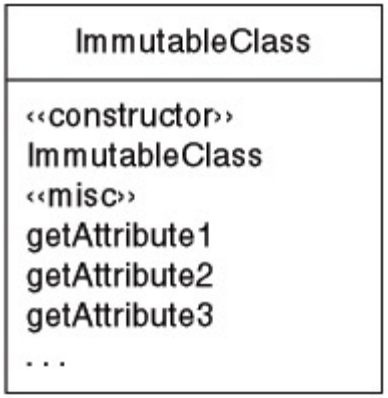
Code example

```
public class Cat {
  private ISoundBehaviour sound = new
MeowSound();

  public void makeSound() {
    this.sound.makeSound();
  }

  public void
setSoundBehaviour(ISoundBehaviour
newsound) {
        this.sound = newsound;
  }
}
```

```
public class Main {
        public static void main(String
args[]) {
                Cat c = new Cat();
                // Delegation
                c.makeSound();//Output: Meow
                // now to change the sound it
makes
                ISoundBehaviour newsound =
new RoarSound();

                c.setSoundBehaviour(newsoun
d);
                // Delegation
                c.makeSound();//Output: Roar
        }
}
```

Immutable

- Intent
  - Increase the robustness of objects that share references to the same object and reduce overhead of concurrent access to an object. It accomplishes this by forbidding any of the object's state information to change after the object is constructed.
- Motivation
  - const-correctness is invaluable.
  - avoid synchronization issues.
- Applicability
  - Use the Immutable pattern when
    - an object isn't suppose to change state during it's lifetime.
- Structure
- Participants
  - Immutable
    - allows modification of state variables only upon creation.
    - provides access methods for exposed state variables.
    - may provide other services.



Immutable

- Consequences
  - Advantages:
    - there is no need to write code to manage state changes.
    - no need to synchronize threads that access immutable objects.
  - Disadvantages:
    - operations that would otherwise have changed the state of an object must create a new object.
- Implementation
    - no method other than the constructor should modify the values of a classes instance variables.
    - any method that computes new state information must store that information in a new instance of the same class, rather then modify the existing object's state.

Immutable

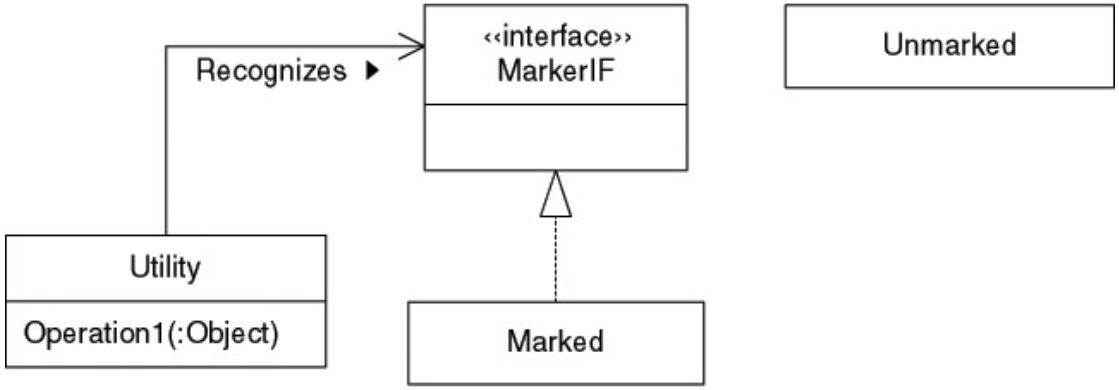
- Related Patterns
  - **Single Threaded Execution.** The Single Threaded Execution pattern is the pattern most frequently used to coordinate access by multiple threads to a shared object. The Immutable Object pattern can be used to avoid the need for the Single Threaded Execution pattern or any other kind of access coordination.
  - **Read-Only Interface.** The Read-Only Interface pattern is an alternative to the Immutable Object pattern. It allows some objects to modify a value object while other objects can only fetch its values.

Marker Interface

- Intent
  - The Marker Interface pattern uses the fact that a class implements an interface to indicate semantic Boolean attributes of the class.
- Motivation
  - unrelated concepts do have something in common
    - Serializable,Cloneable
  - however, how to use this information is context-dependent
- Applicability
  - Use the Marker Interface pattern when
    - as “intent” states - to indicate a semantic attribute.

Marker Interface

- Structure



- Participants
  - MarkerIF
    - marks descendants for a certain mark
  - Marked
    - may be passed to Utility
    - supports some thing it chose to be marked with.
  - Utility
    - queries objects for `MarkerIF` interface and if so uses this information.
  - Unmarked
    - can be passed to Utility but doesn't comply to `MarkerIF` consequences.

Marker Interface

- EqualbyIdentity Example
  - Container objects, such as java.util.ArrayList, call an object's equals method when performing a search of their contents to find an object that is equal to a given object.
  - Such searches might call an object's equals method for each object in the container objects.
  - This is wasteful in those cases where the object being searched for belongs to a class that does not override the equals method.
  - It is faster to use the == operator to determine whether two objects are the same object than it is to call the Object class's implementation of the equals method.
- Collaborations
  - Clients query if a given objects is an instance of the marker interface. This works because inheritance is also sub-typing.

Marker Interface

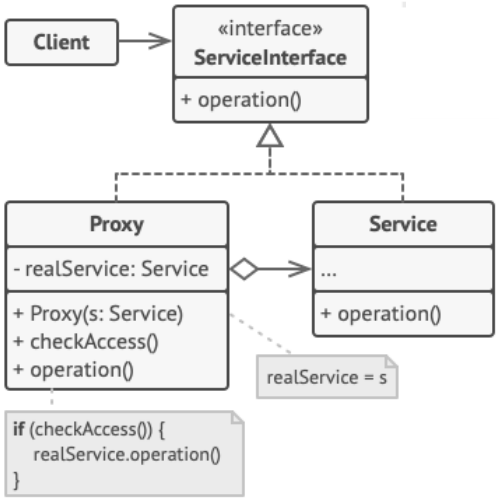
- Code Example

```
public class LinkedList implements Cloneable, java.io.Serializable {
...
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is normally determined by calling the given
     * object's equals method.  However, if the given object implements
     * the EqualByIdentity interface, then equality will be determined
     * by the == operator.
     */
    public LinkedList find(Object target) {
        if (target == null || target instanceof EqualByIdentity)
          return findEq(target);
        else
          return findEquals(target);
    }  // find(Object)
```

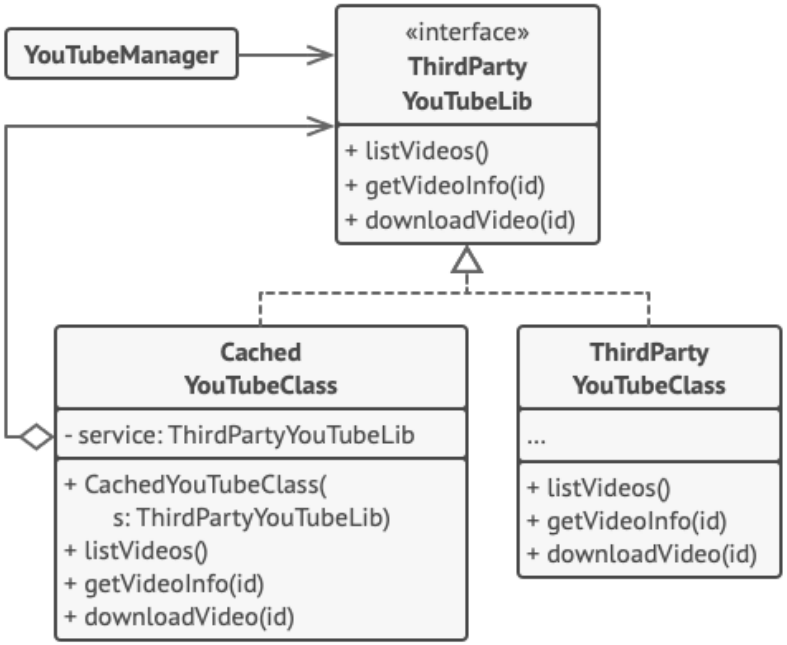
```
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by the == operator.
     */
    private synchronized LinkedList findEq(Object target) {
...
    }  // find(Object)
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by calling the given
     * object's equals method.
     */
    private synchronized LinkedList findEquals(Object target) {
...
    }  // find(Object)
}  // class LinkedList
```

Proxy

- Intent
  - lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



Proxy



- Applicability
  - lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Proxy

- Example uses
  - **Lazy initialization (virtual proxy)**. This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
    - Instead of creating the object when the app launches, you can delay the object's initialization to a time when it's really needed.
  - **Access control (protection proxy)**. This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
    - The proxy can pass the request to the service object only if the client's credentials match some criteria.
  - **Local execution of a remote service (remote proxy)**. This is when the service object is located on a remote server.
    - In this case, the proxy passes the client request over the network, handling all of the nasty details of working with the network.

Proxy

- Example uses
  - **Logging requests (logging proxy)**. This is when you want to keep a history of requests to the service object.
    - The proxy can log each request before passing it to the service.
  - **Caching request results (caching proxy).** This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.
    - The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.
  - **Smart reference**. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.