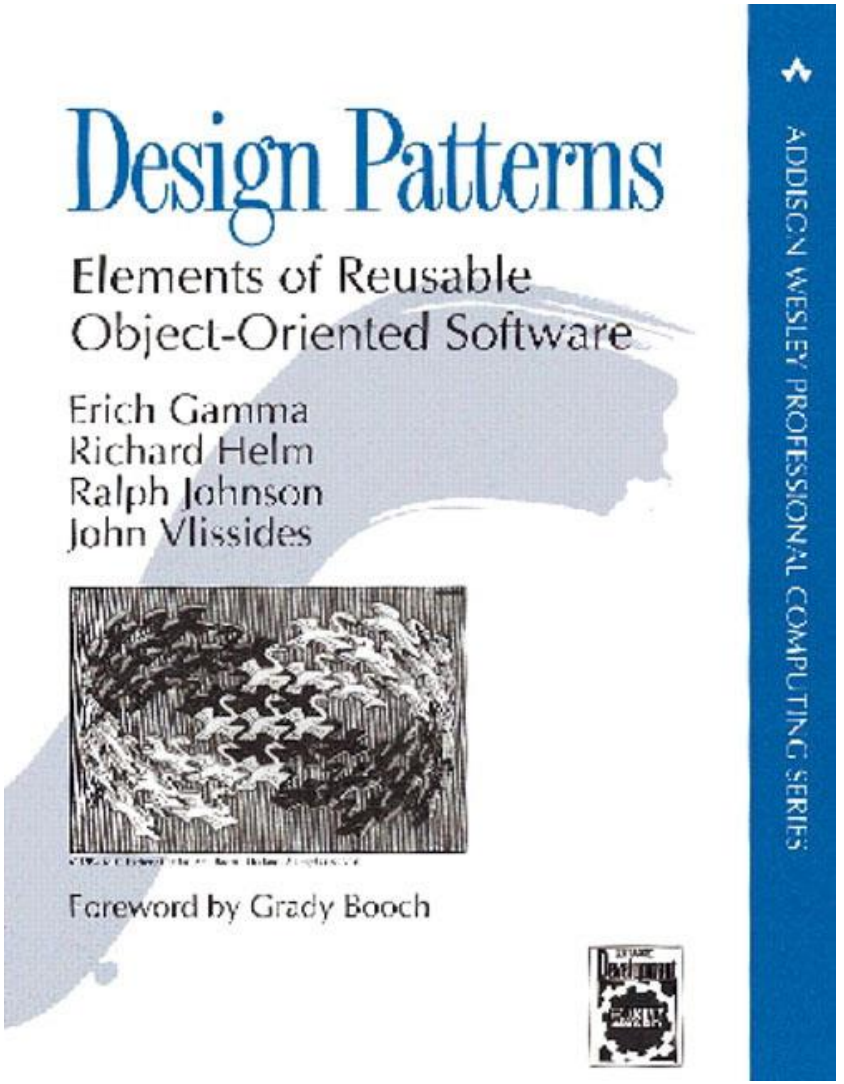
Design patterns

Design patterns: definition(s)

- A design pattern is:
  - A standard solution to a common programming problem
  - A technique for making code more flexible by making it meet certain criteria
  - A design or implementation structure that achieves a particular purpose
  - A high-level programming idiom
  - The shape of an object diagram or object model

Design patterns: classification

- **from DP Bible: GoF (Gang of Four)**
  - **Creational patterns**
  - **Behavioral patterns**
  - **Structural patterns**



Patterns you already know…

- Encapsulation / Data hiding
- Subclassing / Inheritance
- Iterations
- Exceptions

Encapsulation

Encapsulation

- Problem:
  - Exposed fields can be manipulated from outside
- Solution
  - Hide some members, restrict manipulations
- Disadvantages
  - Incomplete interface
  - Reduced performance through indirection

Encapsulation: examples

```
// basic class for drawable components
public abstract class Component {
  private int x;
  private int y;
  private int width;
  private int height;

  public void Draw() {}
  public void Add(Component c){ }
  public void Remove(Component c){ }
  public Component GetChild(int k)
    { return null; }
}
```

Inheritance

Inheritance

- Problem:
  - Similar things have similar members
  - Undesired repetitions
- Solution
  - Inherit default member from a superclass
  - Select desired implementation via runtime dispatching
- Disadvantages
  - Code for a class is spread out

Example: inheritance & polymorphism

```
public abstract class Measurement
     : IComparable, ICloneable
{
   public Measurement()
   {
       EdgeID = "";
   }
   public string SensorID {get;set;}
   public string EdgeID   {get;set;}
   public Point Coordinates {get;set;}
   public long ID {get;set;}
   public DateTime Date {get;set;}
   public double Velocity {get;set;}
   . . .
   public int CompareTo(object obj)
   {
       . . .
   }
   public object Clone()
   {
       . . .
   }
   public virtual string GetUniqueID()
   {
       . . .
   }
}
```

```
public class MSamariter
    : Measurement, IMeasurement,
      I_PostgresObject
{
    public string Code { get; set; }
    public int Dienststelle { get; set; }
    public int GesamtStatus { get; set; }
    public DateTime LastStatus { get; set; }
    public string Status { get; set; }
    public DateTime SysTime { get; set; }

    #region implement I_PostgresObject
    public string GetBulkString()
    {
        . . .
    }
    public void SetBulkString(string _data)
    {
        . . .
    }
    #endregion

    public override string GetUniqueID()
    {
        . . .
    }
}
```

Iteration

Iteration

- Problem:
  - For accessing all members of a collection specialized traversal mechanisms must be implemented
- Solution:
  - Implementations provide traversals
  - Results are communicated via a standard interface
- Disadvantages
  - Iteration order is fixed by implementation.

Exceptions

Exceptions

- Problem:
  - Errors occurring in one part of the code should be handled elsewhere
- Solution:
  - Introduce language structures for throwing and catching errors
- Disadvantages:
  - Hard to detect where an exception will be handled.

Selected patterns

Creational patterns

Creational patterns: types

- Abstract factory
- Builder
- Factory method
- Prototype
- **Singleton**

Singleton

Singleton

- Problem:
  - There should exist only one instance from a specific class
- Solution:
  - Hide operations that create instances
  - Allow creation of and access to the sole instance through a static function
- Disadvantages:
  - Difficult to subclass

Singleton: example

```
public class StickSet{
    private List<Istick> stick;

    private static StickSet instance = null;

    private StickSet() {
          this.sticks = new List<IStick>();
    }
    public static StickSet Instance
    {
          get {
            if (instance==null)
              instance = new StickSet();
            return instance;
          }
    }
    public void SetSticks(List<Istick>
          . . .
    }
}
```

```
public class StickSetClient {
  public static void main (String arg[])
  {
    StickSet.Instance.SetSticks(
         new List<IStick>());
  }
}
```

Structural patterns

Structural patterns

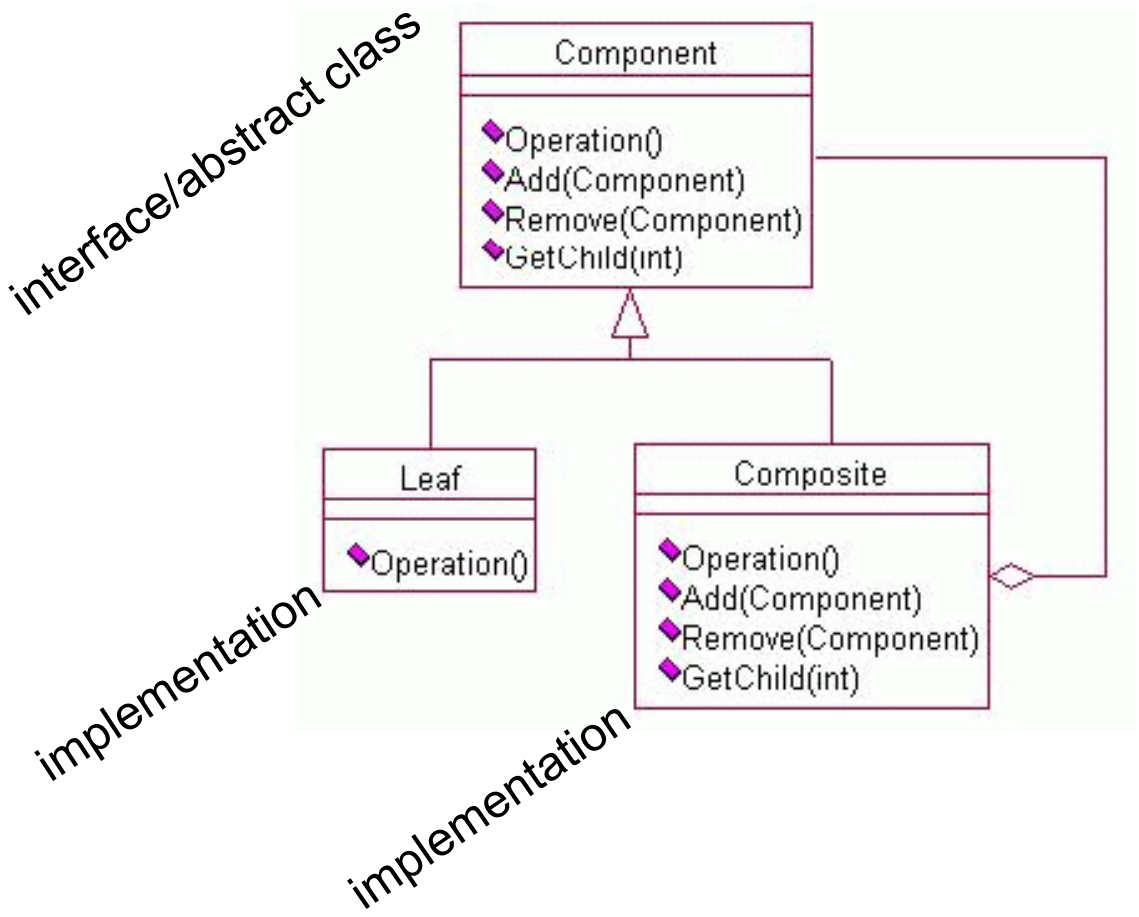
- Adapter
- Bridge
- **Composite**
- Decorator
- Façade
- Flyweight
- Proxy

Composite

Composite

- Problem:
  - Representing hierarchies of objects without differentiating between individual objects and containers (objects that contain other objects)
- Solution
  - Define a common interface for all objects
  - Composite objects forward requests to children, individual objects handle requests directly
- Disadvantages
  - Some operations in the interface have no meaning to individual objects

Composite: object diagram



Composite: example

```
public abstract class Component {
  private int x;
  private int y;
  private int width;
  private int height;
  public void Draw() {}
  public void Add(Component c){ }
  public void Remove(Component c){ }
  public Component GetChild(int k)
    { return null; }
}
```

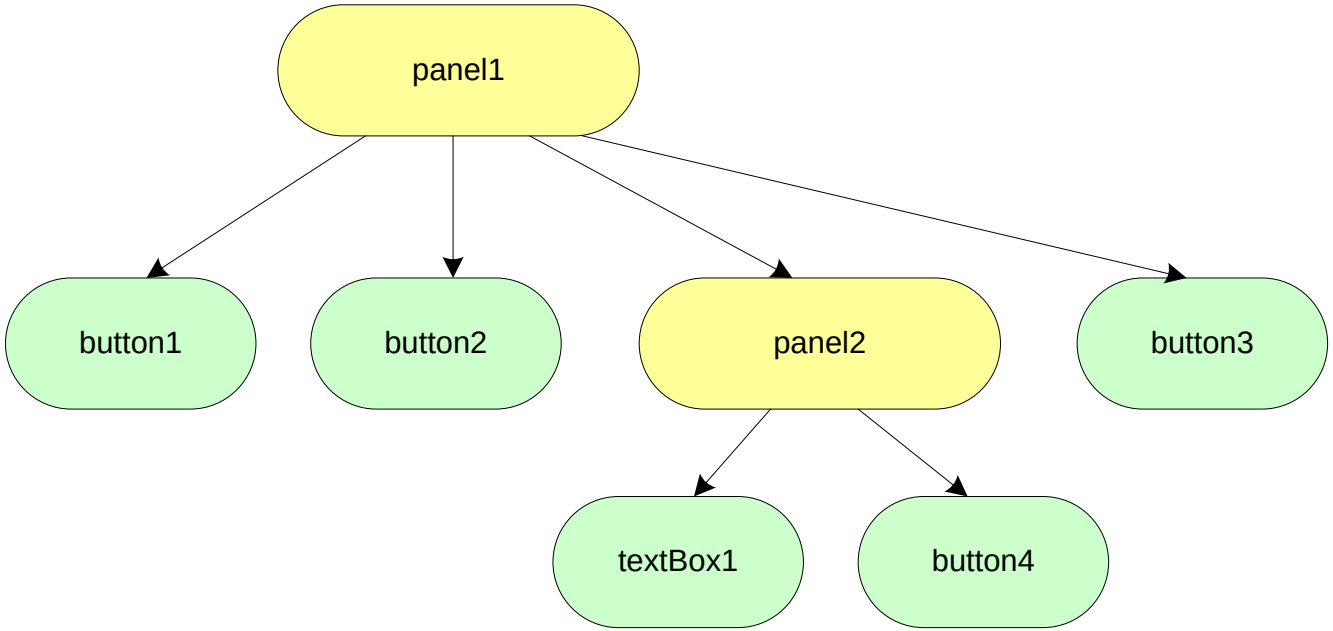
```
public class Button : Component {
  public void Draw() {
    // code for drawing a button
  }
}

public class TextBox : Component {
  public void Draw(){
    // code for drawing a TextBox
  }
}

// the Add / Remove methods
// are meaningless
// for these classes
```

```
public class Panel : Component{
  private List components;
  public void Add(Component c){
    components.Add(c);
  }
  public void Draw() {
    for (int i=0; i<components.size(); i++)
      ((Component)components[i]).Draw();
  }
}
```

Composite: typical object structure



Behavioral patterns

Behavioral patterns

- Chain of responsibility
- **Command**
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Command

Command

- Problem:
  - Similar requests should be supported by unrelated classes
- Solution:
  - Encapsulate a request as an object
- Disadvantages
  - [Decouples classes from operations]

Command: example

```
public interface IComparable {
  public int CompareTo(IComparable c);
}
public interface ISortingAlgorithm {
  public Icomparable[] Array { get; set; }
  public void Sort();
}
```

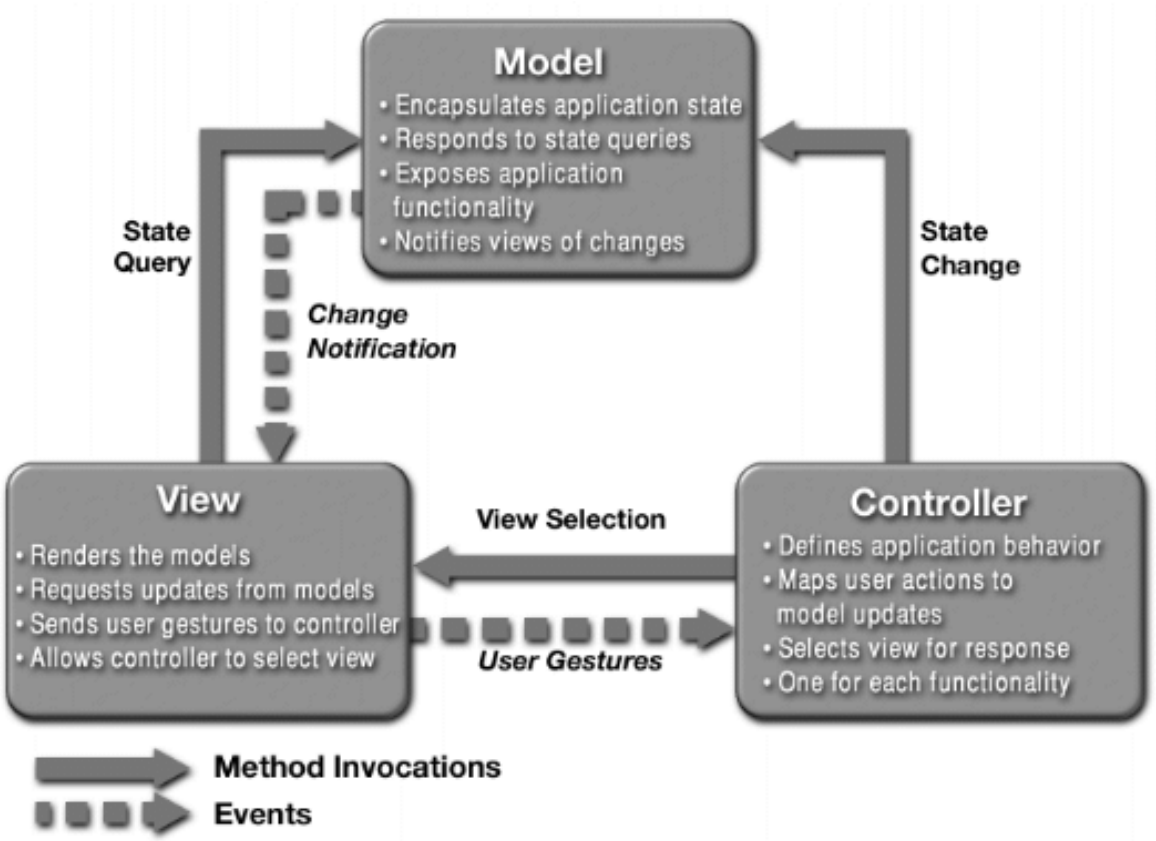
```
public class BubbleSort : ISortingAlgorithm {
  public IComparable[] Array { get; set; }
  public BubbleSort
    (IComparable[] array)
  {
    this.Array = array;
  }
  public void Sort()
  {
    // Implements the bubble-sort algorithm.
    // Before exiting from this function,
    //    this.Array will have the elements
    //    in increasing order.
  }
}
```

Other design patterns:

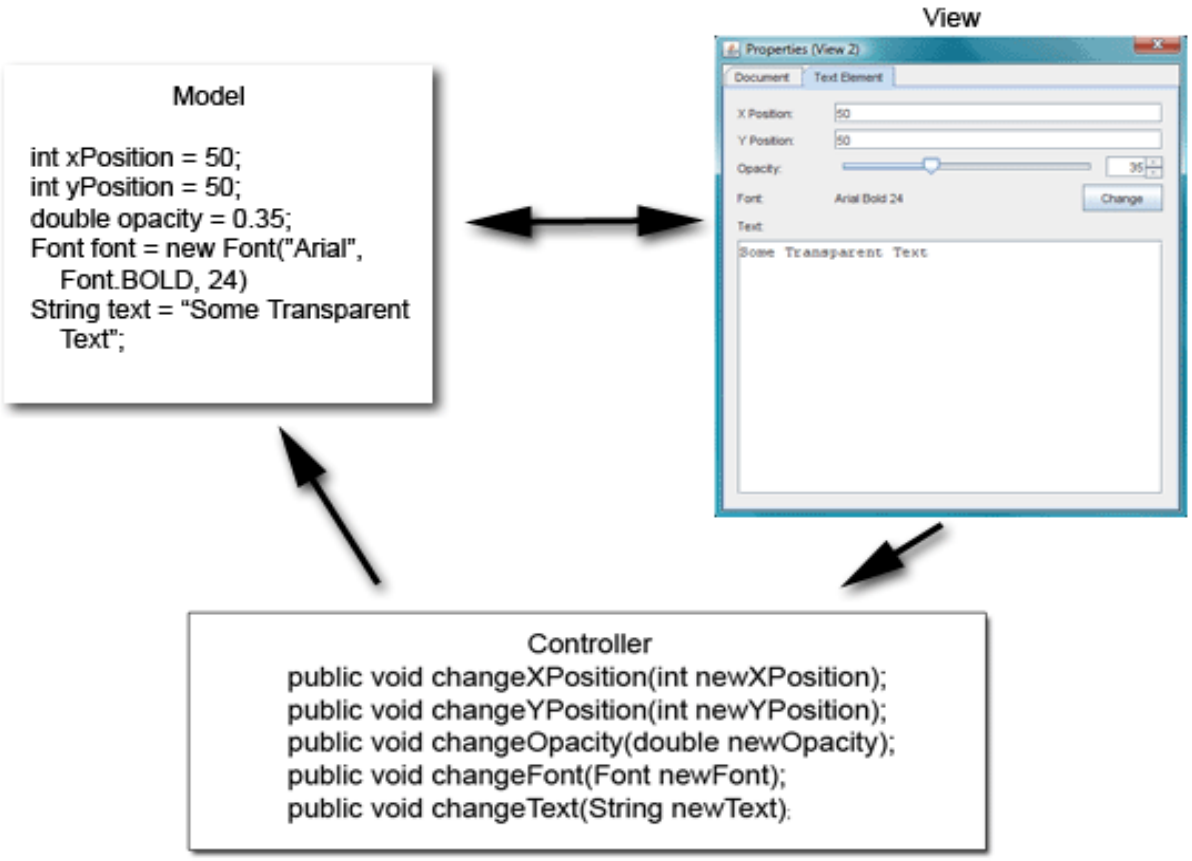
Model-View-Controller

Model-View-Controller

- Problem: isolate application data and logic from input and presentation



Model-View-Controller



Homework

- Can any of the parts in the ADMSys architecture (see slide 20, lecture 6) be implemented as a singleton?
  - What are the disadvantages?