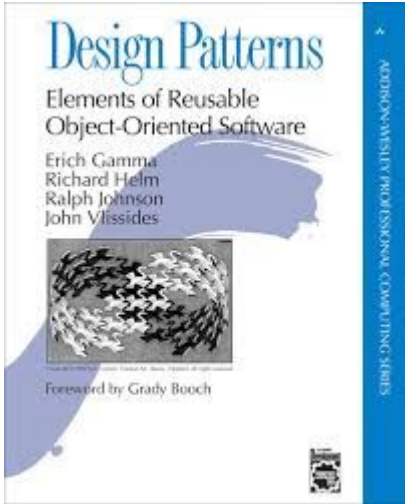
Design Patterns

(and anti-patterns)

Design Patterns

The Gang of Four defined the most common object-oriented patterns used in software.

- These are only the named ones
- Lots more variations exist



Design Patterns

Why are they useful?

- Common terminology
- Lots of example uses
- They promote re-use

Design Patterns

The problem:

- Lots of buzzword sounding terms
- Some patterns very similar to others

Patterns describe the general principle, *not the implementation* (delegation, message passing)

Non-OO Patterns

General

- Cache, Buffer
- Lazy Initialization

Functional

- Partial Application
- Continuation

Pattern Categories

Three categories of patterns exist:

- Creational
  Managing class instances
- Structural
  Relationships between classes (API)
- Behavioral
  Communication between objects

Familiar Patterns

Familiar Patterns

Singleton	Ensure a class has only one instance, and provide a global point of access to it. Easy to use, but introduces dependencies.	Creational
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its listeners are notified.	
Factory	Create objects without specifying their concrete class, decoupling code and adding flexibility. Many ways to implement.	
Visitor	Delegate work to another object, often while iterating over a collection, avoiding switch statements and monolithic code.	

Familiar Patterns

Singleton	Ensure a class has only one instance, and provide a global point of access to it. Easy to use, but introduces dependencies.	
Iterator	Provide a way to access the elements of an array / list / dict sequentially without exposing its underlying representation.	Behavioral
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its listeners are notified.	
Factory	Create objects without specifying their concrete class, decoupling code and adding flexibility. Many ways to implement.	
Visitor	Delegate work to another object, often while iterating over a collection, avoiding switch statements and monolithic code.	

Familiar Patterns

Singleton	Ensure a class has only one instance, and provide a global point of access to it. Easy to use, but introduces dependencies.	
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its listeners are notified.	Behavioral
Factory	Create objects without specifying their concrete class,decoupling code and adding flexibility. Many ways to implement.	
Visitor	Delegate work to another object, often while iterating over a collection, avoiding switch statements and monolithic code.	

Familiar Patterns

Singleton	Ensure a class has only one instance, and provide a global point of access to it. Easy to use, but introduces dependencies.	
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its listeners are notified.	
Factory	Create objects without specifying their concrete class, decoupling code and adding flexibility. Many ways to implement.	Creational
Visitor	Delegate work to another object, often while iterating over a collection, avoiding switch statements and monolithic code.	

Familiar Patterns

Singleton	Ensure a class has only one instance, and provide a global point of access to it. Easy to use, but introduces dependencies.	
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its listeners are notified.	
Factory	Create objects without specifying their concrete class, decoupling code and adding flexibility. Many ways to implement.	
Visitor	Delegate work to another object, often while iterating over a collection, avoiding switch statements and monolithic code.	Behavioral

Familiar Patterns

Singleton

```
EventManager.Instance.EventQueue();
```

Iterator

```
while (iter.moveNext()) {
    Console.WriteLine(iter.Current());
}
```

Observer

```
widget.AddListener(this, “OnClick”);
```

Factory

```
newWidget = WidgetFactory.Create(WidgetType.Button);
```

Visitor

```
for (var visitor : objects) {
    visitor.printOn(this);
}
```

Familiar Patterns

Singleton

```
EventManager.Instance.EventQueue();
```

Iterator

```
while (iter.moveNext()) {
    Console.WriteLine(iter.Current());
}
```

Observer

```
widget.AddListener(this, “OnClick”);
```

Factory

```
newWidget = WidgetFactory.Create(WidgetType.Button);
```

Visitor

```
for (var visitor : objects) {
    visitor.printOn(this);
}
```

Familiar Patterns

Singleton

```
EventManager.Instance.EventQueue();
```

Iterator

```
while (iter.moveNext()) {
    Console.WriteLine(iter.Current());
}
```

Observer

```
widget.AddListener(this, “OnClick”);
```

Factory

```
newWidget = WidgetFactory.Create(WidgetType.Button);
```

Visitor

```
for (var visitor : objects) {
    visitor.printOn(this);
}
```

Familiar Patterns

Singleton

```
EventManager.Instance.EventQueue();
```

Iterator

```
while (iter.moveNext()) {
    Console.WriteLine(iter.Current());
}
```

Observer

```
widget.AddListener(this, “OnClick”);
```

Factory

```
newWidget = WidgetFactory.Create(WidgetType.Button);
```

Visitor

```
for (var visitor : objects) {
    visitor.printOn(this);
}
```

Familiar Patterns

Singleton

```
EventManager.Instance.EventQueue();
```

Iterator

```
while (iter.moveNext()) {
    Console.WriteLine(iter.Current());
}
```

Observer

```
widget.AddListener(this, “OnClick”);
```

Factory

```
newWidget = WidgetFactory.Create(WidgetType.Button);
```

Visitor

```
for (var visitor : objects) {
    visitor.printOn(this);  // Double-dispatch (delegate)
}
```

Familiar Patterns [Extended]

Null Object Abstract the handling of null away from the client by introducing Null objects or Null subclasses. This pattern should be used carefully as it can make errors/bugs appear as normal program execution.

Object Pool Create and re-use objects instead of creating and destroying them. Speeds up use and reduces garbage. [UITableView dequeueReusableCellWithIdentifier:]

Familiar Patterns, Unfamiliar Names

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An abstract class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, and for logging, access control, etc.

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An abstract class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, and for logging, access control, etc.

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An abstract class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, and for logging, access control, etc.

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An **abstract** class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, and for logging, access control, etc.

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An abstract class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, and for logging, access control, etc.

Familiar Patterns, Unfamiliar Names

Builder	Separate the construction of complex objects from its representation.
Chain of Responsibility	Avoid coupling by allowing a chain of objects to handle the request. The search is over when one objects reports that it has handled the request.
Facade	Wraps a complicated subsystem with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper in our code.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An abstract class that needs “placeholder” methods filled in.
Prototype	Create new objects by cloning existing ones. Makes it easy to create templates at runtime. Similar to the Factory pattern.
Proxy	Provide a placeholder for another object to control access to it. Used to make remote objects look local, for logging, access control, etc.

Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations. Related to the Factory Pattern.

```
public class AnimatorBuilder {
   public AnimationBuilder Rotate(float angle) { ... }
   public AnimationBuilder Translate(float x, float y) { ... }
}

Animation animation = new AnimatorBuilder().Rotate(45).Translate(10, 10).Build();
```

Chain of Responsibility Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations. Example: [view nextResponder]

```
public class Widget {
   public void SendEvent(Event event) {
       …
       while (!widget.HandleEvent(event)) {
           widget = widget.NextResponder;
       }
   }
}
```

Facade Pattern

Wraps a complicated system with a simpler interface. Also helps abstract APIs such as Facebook. Commonly seen as *Helper classes in our code.

```
public class LoginHelper {
   public static void Login() {
       // Do a bunch of complicated stuff here
       // Internal implementation can change
   }
}

LoginHelper.Login();  // Implementation and complexity is hidden
```

Template Method Pattern

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. An **abstract class** that needs “placeholder” methods filled in. Quintessential “OO”.

```
public abstract class Widget {
    public void Draw() {
        this.DrawRect(this.Bounds);
    }

    public virtual void DrawRect(Rect rect);  // Override this method to do something useful
}


widget.Draw();
```

Prototype Pattern

Create new objects by cloning existing ones. Makes it easy to create templates at runtime and provides more flexibility. Similar to the Factory pattern.

```
public class Widget {
   public Widget Clone() {
        Widget widget = new Widget();
        widget.CopyPropertiesFrom(self);

        return widget;
   }
}

LabelWidget newLabel = titleLabel.Clone();
```

Proxy Pattern

Provide a placeholder for another object to control access to it. Makes remote objects look local, or can be used as a conduit for logging or access control. Similar to Decorator, but **does not add or enhance** functionality. Implemented via delegation or reflection.

```
interface ICanvas { void DrawLine(float x, float y); }

class CanvasLoggingProxy : ICanvas {
    private ICanvas canvas;

    public void DrawLine(float x, float y) {
        this.canvas.DrawLine(x, y);
    }
}
```

Fuzzy/Unclear Patterns

Fuzzy/Unclear

Adapter
Convert the interface of a class into another interface clients expect.

Bridge
Use interfaces instead of abstract classes to decouple interface from impl.

Composite
Allows clients treat individual objects and compositions of objects uniformly.

Decorator
Wrap an object to provide additional features, using the same interface.

Flyweight
Use sharing to support large numbers of fine-grained objects efficiently.

Private Class Data
Encapsulate class data initialization, separating data from methods that use it.

Command
Encapsulate a function call parameter as an object, enabling logging, undo, oo-callbacks, etc. aka Action.

Interpreter
Execute a sequence of commands, enabling dynamic execution, batching.

Mediator
Design an intermediary to decouple many peers, which promotes loose coupling.

Memento
Allow an object to restore itself back to its previous state (e.g. “undo”).

State
Allow an object to alter its behavior when its internal state changes.

Strategy
Define a family of algorithms, encapsulate each one, and make them interchangeable.

Adapter Pattern

Convert the interface of a class into another interface clients expect. Used to encapsulate glue code.

Related to the Facade pattern, but much smaller in scope: System vs class. Iterator uses Adapter.

```
public interface IIndexable<T> {
   T ObjectAtIndex(int index);
}

public class ArrayAdapter<T> : IIndexable {
   private array[T];

   public T ObjectAtIndex(int index) {
       return self.array[index];
   }
}
```

Bridge Pattern

Use interface inheritance to decouple interface from implementation. Avoids inheritance explosion by not subclassing and only inheriting interfaces. Flexible, but may bloat code because of non re-use.

```
interface IStreamReader { public char Read(); }
interface IStreamWriter { public void Write(char c); }

class ReadWriteStream : IStreamReader, IStreamWriter {
    public char Read() { … }
    public void Write(char c) { … }
}

~~class WriteStream : ReadStream { … }~~
~~class ReadWriteStream : WriteStream { … }~~
```

Composite Pattern

Allows clients treat individual objects and compositions of objects uniformly. Examples: UIView - Views with children treated the same as views with no children. Files and folders in Finder / Explorer.

```
public interface Node {
    public int Name();
}

public class File : Node {
    public int Name() { … }
}

public class Folder : Node {
    private Node[] children;

    public int Name() { … }
}
```

Decorator Pattern

Wrap an object to provide additional features, using the same interface. Alternative to subclassing.

Related: Proxy Pattern, Delegation. Adds or extends an object, vs Proxy which intercepts.

```
interface IDrawable {
   public void Draw();
}

class BackgroundDecorator : IDrawable {
   private IDrawable drawable;

   public void Draw() {
       this.DrawBackground(), this.Drawable.Draw();
   }
}

canvas.addDrawable(new BackgroundDecorator(new Drawable()));
```

Flyweight Pattern

Use sharing to support large numbers of fine-grained objects efficiently. Process vs thread, view vs grid cell: View draws cells, instead of a View per cell. Saves memory, possibly more performant.

```
class GridCell {
   public void drawOnView(Widget widget) { }
}

class GridView : Widget {
   List<GridCell> cells;

   foreach (cell in this.cells) {
       cell.drawOnView(this);  // Also using visitor pattern here
   }
}
```

Private Data Class Pattern

Encapsulate class data initialization, separating data from methods that use it. Used to protect like “protected”, but within the class itself. Can be thought of as a non-sharable “model” class.

```
class Customer {
   private struct Data {
       private string name;
       public string Name() { return name; }
   }

   private data;

   public Customer(string name) {
       this.data = Data() { name };
   }
}
```

Fuzzy/Unclear/Rare Patterns Part 2

Command Pattern

Encapsulate a function call parameter as an object, enabling logging, undo, callbacks, etc. aka Action.

Combine with other patterns: Command + Interpreter = Macro.

Related: Closure (functional equivalent)

```
public class RotateImageAction {
   private float angle;

   public RotateImageAction(float angle) { this.angle = angle; }

   public execute() { … }
}

image.executeAction(new RotateImageAction(45));
```

Interpreter Pattern

Execute a sequence of commands, enabling dynamic execution, batching. Can be a syntactic language or simply an Abstract Syntax Tree, a model representing syntax structure.

Related: Command Pattern, Composite Pattern

```
List<IAction> actions = new List<Action>() {
   new RotateImageAction(angle: 45),
   new BlurImageAction(blur: 0.5),
   new ScaleImageAction(x: 2.0, y: 2.0)
}

image.executeActions(actions);
```

Mediator Pattern

Design an intermediary to decouple many peers, which promotes loose coupling by keeping objects from referring to each other explicitly. Example: ViewController in between code and the view it controls.

```
public class Mediator {
    public void Push();
    public void Pop();
}

public class Producer {
    private Mediator mediator;
    public void Produce() { mediator.Push(new Item()); }
}

public class Consumer {
    private Mediator mediator;
    public void Consume() { Item item = mediator.Pop(); }
}
```

Memento Pattern

Allow an object to restore itself back to its previous state without violating encapsulation (e.g. “undo” or “rollback” operations). The memento is **opaque** to the outside, and must not operated on.

```
public class ButtonWidget {
    private ButtonState currentState;

    public void SetState(OpaqueHandle state) {
        this.currentState = (ButtonState)state;
    }

    public OpaqueHandle GetState() { }
}
```

State Pattern

Allow an object to alter its behavior when its internal state changes. Can be implemented as a delagate.

Simply an Object Oriented state machine (possibly using the Visitor pattern to delegate).

Related: Strategy

```
class CircleState : IShapeState {
   public Draw(Shape shape) { … }
}

class Shape {
   private IShapeState shapeState;

   public void Draw() { this.shapeState.Draw(this); }  // Also using visitor pattern here
}

shape.SetState(circleState);
shape.Draw();
```

Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. **Uses interfaces**. Related to Template Method.

```
interface IImageEncoder {
   public Buffer encode(Buffer rawData) { … }
}

class Image {
   private Buffer imageData;

   public Save(string fileName, IImageEncoder encoder) {
       IO.Open(fileName).Write(encoder.encode(this.imageData));
   }
}

image.Save(“image.png”, new PngEncoder());
```

Pattern Similarities

The strategy pattern allows you to change the implementation of something used at runtime. The Decorator pattern allows you augment (or add) existing functionality with additional functionality at run time. Strategy uses **interfaces**, while Template Method uses **abstract** classes.

The difference between Strategy and State is in **binding times**: Strategy is a bind-once pattern, whereas State is more dynamic. A change in the state causes it to select from its "palette" of Strategy objects.

Iterator and Adapter are similar, but Adapter is a structural pattern while Iterator is a behavioral pattern.

The Facade only exposes the existing functionality from a different perspective. The Mediator "adds" functionality because it combines different existing functionality to create a new one.

Pattern Similarities

Bridge: (A structural pattern)

Bridge pattern decouples abstraction and implementation and allows both to vary independently. Use this pattern when :

1. Abstractions and implementations have not been decided at compile time
2. Abstractions and implementations should be changed independently
3. Changes in implementation of abstraction should not affect caller application
4. Client should be insulated from implementation details.

Strategy: (Behavioural pattern)

Strategy patterns enable you to switch between multiple algorithms from a family of algorithms at run time. Use Strategy pattern when :

1. Multiple versions of algorithms are required
2. The behaviour of class has to be changed dynamically at run time
3. Avoid conditional statements

Dependency Injection

Pass around objects instead of using singletons or direct access to prevent tight coupling.
Implements **inversion of control** for resolving dependencies.

```
public class Foo {
    public void doWork() {
        EventManager.Instance.EventQueue();  // Accesses concrete class EventManager directly
    }
}
```

Using Dependency Injection:

```
public class Foo {
    IEventManager eventManager;

    public void doWork() {
        eventManager.EventQueue();  // Not coupled (using interface), easier to unit test
    }
}
```

FIN