# CSE 3902: Design Patterns

Justin Holewinski

The Ohio State University

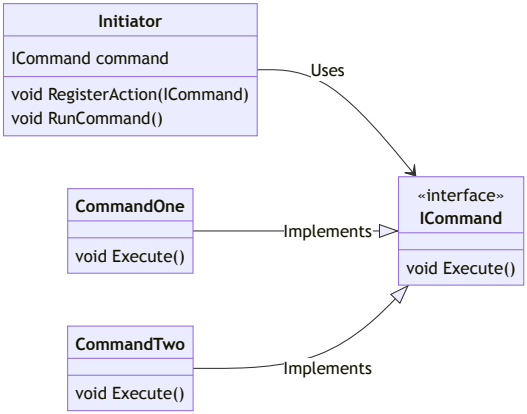
# Command Pattern

Decouple action from initiator

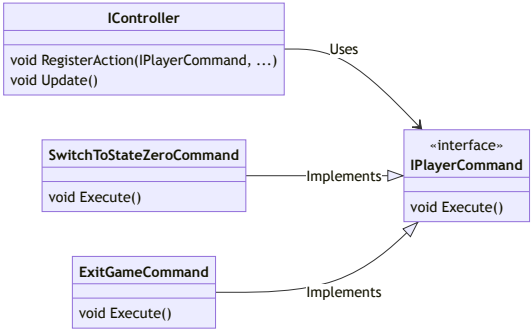
- Initiator just sees generic interface
- User can implement multiple actions without changing initiator

Example

- Imagine `PlayerAttack` is a command implementation
- Who might need to know about `PlayerAttack`?



# Command Pattern Example



# Command Pattern Example

```
public interface IPlayerCommand
{
    public void Execute(MyGame game);
}

public class PlayerExitCommand : IPlayerCommand
{
    public void Execute(MyGame game)
    {
        game.ConfirmExit();
    }
}
```

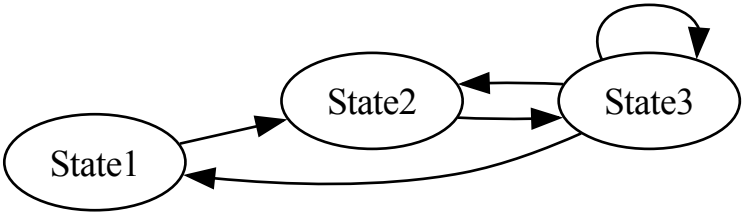
```
public class KeyboardController : IController
{
    private Dictionary<Keys,
        IPlayerCommand> commands;

    public void Update()
    {
        if (/* key pressed */)
        {
            commands[key].Execute();
        }
    }
}
```

# State Machine

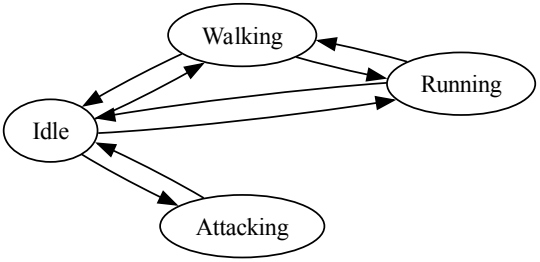
An object can be in one of a finite set of states

The state machine defines the possible set of transitions



# State Machine Example

Player States:




- Do we need the transitions between Walking and Running?
- What would change if we added a Jumping state?
- What would change if we wanted to run and attack at the same time?

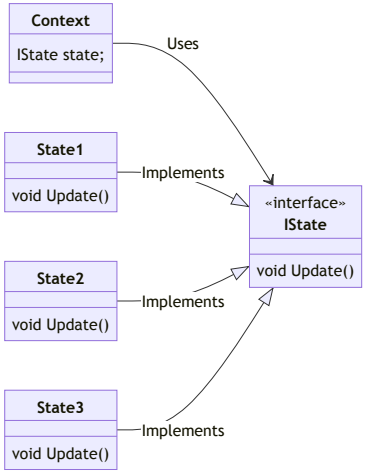
# State Machine Example

```
public class Player {
    private PlayerStateMachine stateMachine;
    public void Attack() {
        stateMachine.Attack();
    }
    public void Jump() {
        stateMachine.Jump();
    }
    public void MoveLeft() {
        stateMachine.MoveLeft();
    }
    public void MoveRight() {
        stateMachine.MoveRight();
    }
}
```

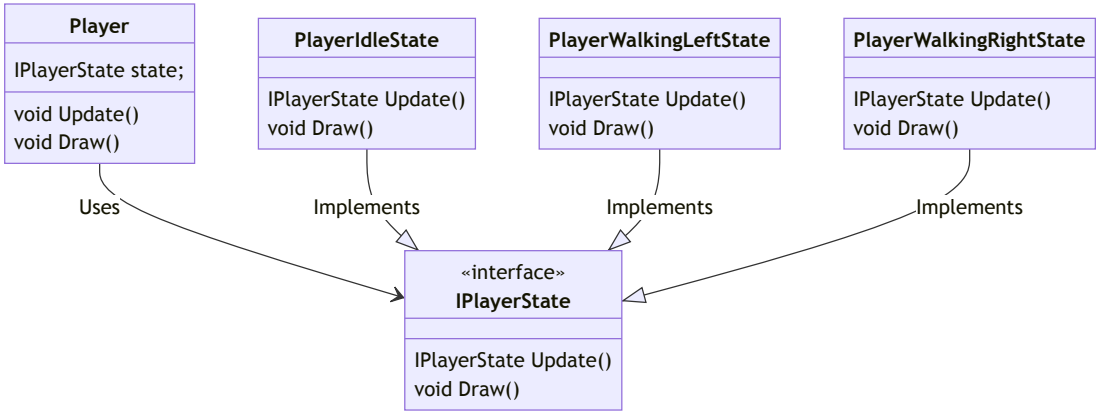
```
public class PlayerStateMachine {
    private enum State {
        IDLE, ATTACKING, JUMPING, MOVING };
    private State state;
    public void Attack() {
        if (state == State.ATTACKING)
            return;
        if (state == State.JUMPING)
            return;
        // Logic for starting an attack
        state = State.ATTACKING;
    }
    public void Jump() {
        if (state == State.JUMPING)
            return;
        if (state == State.ATTACKING)
            return;
        // Logic for starting a jump
        state = State.JUMPING;
    }
}
```

# State Pattern

- Separate state-agnostic and state-specific data and logic
- `Context` object has "states" through `IState` field
- State-specific logic contained in `IState` implementations
- Only one state is "active" at a time
- States define possible transitions

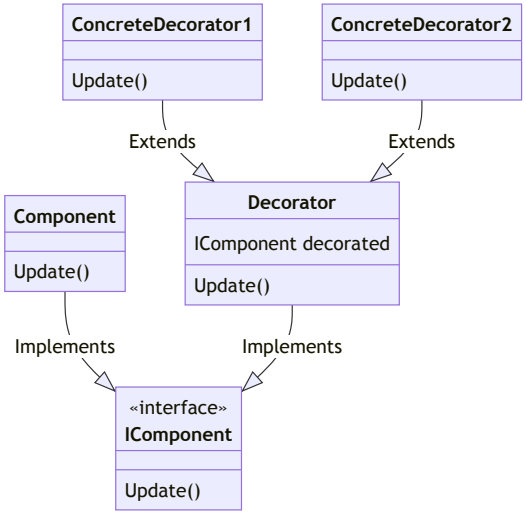


# State Pattern Example



# Decorator Pattern

- *Component* is the base implementation
- Each *decorator* adds new functionality
- Decorators can be added dynamically



# Decorator Pattern

```
public class Decorator : IComponent {
    private IComponent decorated;
    public Decorator(IComponent parent) {
        this.decorated = parent;
    }
    public void Update() {
        decorated.Update();
    }
}

public class ConcreteDecorator1 : Decorator {
    public Decorator(IComponent parent)
        : base(parent) {
        this.decorated = parent;
    }
    public void Update() {
        // Logic specific to decorated object
        SomeSpecialLogic();
        base.Update();
    }
}
```

```
public class Component : IComponent {
    public Component() {}
    public void Update() {
        // Normal component logic
        BasicLogic();
    }
}

public class Context {
    public void Run() {
        IComponent component = new Component();
        component =
            new ConcreteDecorator1(component);

        // What happens in the Update() call below?
        //
        //   ConcreteDecorator1.SomeSpecialLogic()
        //   Component.BasicLogic()
        component.Update();
    }
}
```

# Decorator/State Combination

```
public class Player : IPlayer {
    private IPlayerState state;
    private Game1 game;

    public void Heal() {
        state.Heal();
    }
    public void TakeDamage() {
        if (state.TakeDamage()) {
            game.Player =
                new DamagedPlayer(this, game);
        }
    }
    public void Update() {
        state.Update();
    }
    public void Draw() {
        state.Draw()
    }
}
```

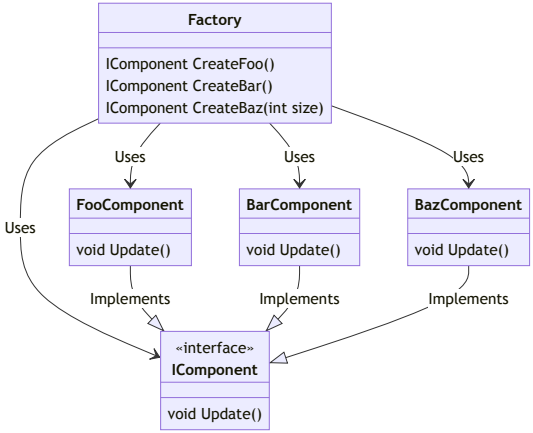
```
public class DamagedPlayer : IPlayer {
    private IPlayer decorated;
    private Game1 game;

    public DamagedPlayer(IPlayer parent,
                         Game1 game) {
        this.decorated = parent;
        this.game = game;
    }
    public void Heal() {
        game.Player = decorated;
    }
    public void Update() {
        decorated.Update();
    }
    public void Draw() {
        decorated.SetDamagedColor();
        decorated.Draw();
    }
}
```

These are examples, not fully functional code blocks!

# Factory Pattern

- Class for creating object instances
- Hides internal details about object creation
- Helps decouple implementations by hiding choice of implementation class



# Factory Pattern

Caller does not care or even see the implementation class, or the details of how to create it:

```
public interface IComponent { /* ... */ }

public class FooComponent : IComponent {
    FooComponent(int startingValue) { /* ... */ }
}

public class Factory {
    private int startingValue = 10;
    public IComponent CreateFoo() {
        return new FooComponent(startingValue);
    }
}
```

# Factory Pattern

Factories can create objects by arbitrary *key*:

```
public interface IComponent { /* ... */ }

public class Factory {
    public IComponent CreateComponent(string name) {
        if (name == "foo") {
            return new FooComponent();
        } else if (name == "bar") {
            return new BarComponent();
        } else if (name == "baz") {
            return new BazComponent();
        } else {
            throw new Exception("Unknown type");
        }
    }
}
```

# Factory Example

```
public interface ISprite { /* ... */ }

public class Factory {
    private Texture2D spriteSheetTexture;
    private const int MarioStartFrame = 10;
    private const int MarioEndFrame = 20;
    // ...

    public ISprite CreateMario() {
        return new Sprite(spriteSheetTexture,
                          MarioStartFrame,
                          MarioEndFrame,
                          MarioWidth,
                          MarioHeight);
    }

    public ISprite CreateGoomba() {
        return new Sprite(/* ... */);
    }
}
```

# Singleton Pattern

A class that can only ever have one instance

```
public class Foo {
    private static Foo instance;

    private Foo() { /* ... */ }

    public static Foo Instance {
        get {
            if (instance == null) {
                instance = new Foo();
            }
            return instance;
        }
    }

    /* Implementation */
}
```

# Singleton Pattern

## Lazy Initialization

```
public class Foo {
    private static Foo instance;

    private Foo() { /* ... */ }

    public static Foo Instance {
        get {
            if (instance == null) {
                instance = new Foo();
            }
            return instance;
        }
    }

    /* Implementation */
}
```

## Eager Initialization

```
public class Foo {
    private static Foo instance = new Foo();

    private Foo() { /* ... */ }

    public static Foo Instance {
        get {
            return instance;
        }
    }

    /* Implementation */
}
```

What are the trade-offs?

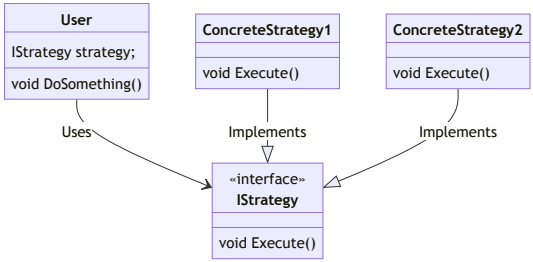
# Singleton Pattern

## A Word of Warning

Making a class a singleton is hard to *undo* due to all of the eventual `Foo.Instance` expressions that will show up in your code. Be very sure you will only ever want one instance of a particular class before making it a singleton!

# Strategy Pattern

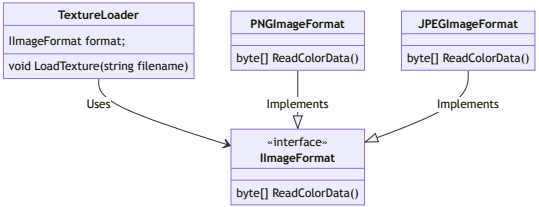
- Delegate choice of algorithm to runtime
- User can select algorithm based on input
- Can be extended with new algorithms without much modification to users



# Strategy Pattern

```
public class TextureLoader {
    private IImageFormat format;
    private byte[] data;

    public void LoadTexture(string filename) {
        string ext = Path.GetExtension(filename);
        switch (ext) {
            case "png":
                format = new PNGImageFormat();
                break;
            case "jpg":
                format = new JPEGImageFormat();
                break;
            default:
                throw new Exception("Bad format");
        }
        data = format.ReadColorData();
    }
}
```



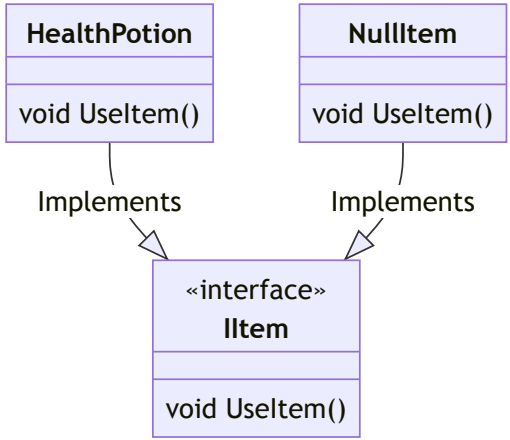
# Null Pattern

- Use special implementation for “null” objects
- Prevents excessive null checks in code
- Null objects normally have empty bodies

```
var item = Player.GetActiveItem();

// With null pattern
item.UseItem();

// Without null pattern
if (item)
{
    item.UseItem();
}
```



# Flyweight Pattern

*Observation*: Classes with many instances often duplicate data

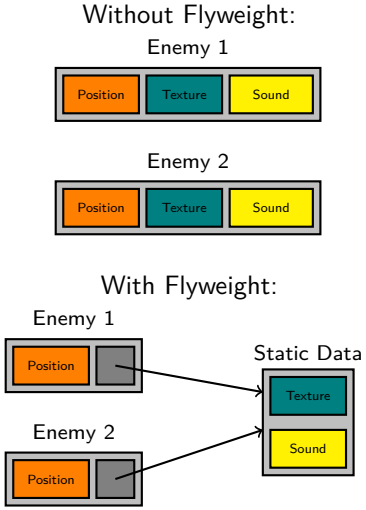
- Fields may have common values across instances
- Object colors, textures, source rectangles

*Flyweight Pattern*: Split class based on whether a field is unique to a particular instance

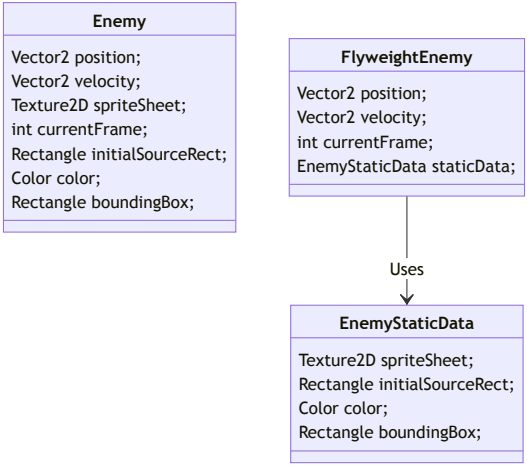
- Position is unique to an object
- Texture may be shared across many instances

Many instances can share the same common data object

- Storing a reference to common data uses less memory than duplicating the data
- Many enemies will share the same sprite sheet
- Many environment blocks will share the same sprite



# Flyweight Pattern



# Object Pools

Maintain a set of objects, called a pool

- "Create" an object by taking one from the pool
- "Delete" an object by returning it to the pool

Pool can be expanding or fixed-size

- Expanding pool creates new instances as needed
- Fixed-size pool has maximize size

Advantages

- Prevent overhead of repeated object creation
  - Calling new can be expensive!
  - Continually creating objects and then dropping references to them makes a lot of work for the garbage collector!
- Place limit on number of object instances

```
public IProjectile CreateProjectile() {
    if (Pool.Count > 0) {
        // We have existing items in the pool, so
        // select the first, remove it from the pool
        // and return it.
        var projectile = Pool[0];
        Pool.RemoveAt(0);
        return projectile;
    }

    // There are no free objects in the pool, so
    // create a new one and return it.
    var projectile = new Projectile();
    return projectile;
}

public void DeleteProjectile(IProjectile projectile) {
    // Reset projectile to default state and add it to
    // the pool.
    projectile.Reset();
    Pool.Add(projectile);
}
```

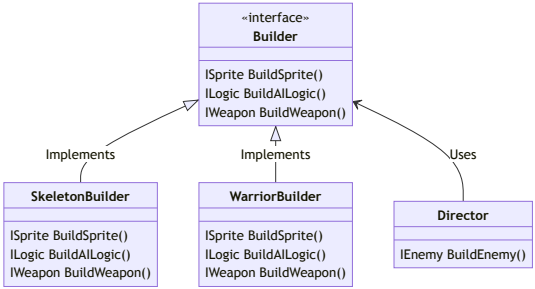
# Builder Pattern

Separate creation of aggregate from creation of parts

- *Builder* defines how individual parts are created
- *Director* defines how whole object is built

Same director can be used with multiple builders

- Construct different objects by swapping out the builder
- Example: each enemy may be constructed in the same way, but with different components



# Memento Pattern

Save a *memento* to allow returning back to the previous state of an object

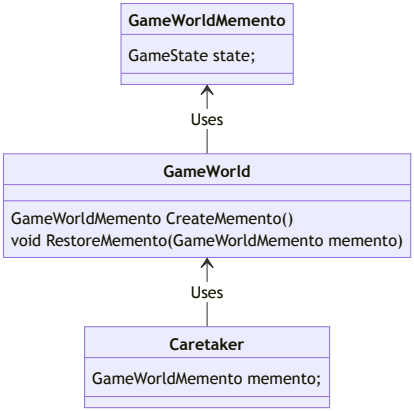
- Capture all state necessary to reconstruct an object

Client (*caretaker*) requests memento from *originator*

- Only originator is allowed to inspect memento state
- Client cannot look into memento, just pass it around

Useful for save game files

- Save Game: Create memento and write to disk
- Load Game: Load memento from disk and restore game objects



# Anti-Patterns

A *design pattern* is a common, well-understood, effective solution to a common software problem.

An *anti-pattern* is a common but *ineffective* solution to a common software problem.

- Often leads to maintenance issues
- May appear attractive at first, but not effective in the long term

Similar to a *code smell*, but at the design level.

Examples:

- *God Class*: one class to rule them all
- *Poltergeists*: wrapper classes that do no actual work