Abstraction and Architecture:

A steady descent into madness

@pseale

What you need to know about me

1. @pseale on twitter
2. You can make this talk better

This is a low-level, detail-oriented talk about

~ ~ Architecture ~ ~

Because I'm working with a specific example, I may skip over the 'why' reasoning – feel free to interject.

As with all architecture, this talk contains

~ ~ Assumptions ~ ~

Long-term developer speed is my priority. Ignore performance*.

Today only, we are banning the following

~ Architecture swear words ~

- Bad/Good
- Elegant
- Testable
- Maintainable
- Robust
- Extensible
- Cohesion/coupling
- SOLID
- DRY
- Spaghetti code
- “I can have you fired”

~ Each abstraction must justify itself ~

THE PLAY

A architectural tragedy in five* acts

Act 1: **Bliss**

Code is **exactly and only** what is **minimally necessary** to make our program work

~ demo ~

ACT I: Bliss

```
protected override void Update(GameTime gameTime) {
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back
      == ButtonState.Pressed || Keyboard.GetState()
      .IsKeyDown(Keys.Escape))
    Exit();

  var keyboardState = Keyboard.GetState();
  _facingDirection = new Vector2(0f, 0f);

  _moveDirection = new Point();
  if (keyboardState.IsKeyDown(Keys.Up))
    _moveDirection.Y--;
  if (keyboardState.IsKeyDown(Keys.Down))
    _moveDirection.Y++;
  if (keyboardState.IsKeyDown(Keys.Left))
    _moveDirection.X--;
  if (keyboardState.IsKeyDown(Keys.Right))
```

ACT I: Bliss

```
  _playerPosition = _playerPosition + _moveDirection;

  var mouseState = Mouse.GetState();
  _firing = mouseState.LeftButton == ButtonState.Press

  var x = Math.Max(Math.Min(mouseState.Position.X, Scr
  var y = Math.Max(Math.Min(mouseState.Position.Y, Scr

  _facingDirection = new Vector2(0f, 0f);
  int xPositionOnScreen = (WidthMidpoint + (_playerPos
  int yPositionOnScreen = (HeightMidpoint + (_playerPo
  _facingDirection.X = ((float)(x - xPositionOnScreen)
  _facingDirection.Y = ((float)(y - yPositionOnScreen)
  float div = 1f/(float)Math.Sqrt(_facingDirection.X*_
_facingDirection.Y*_facingDirection.Y);
  _facingDirection = new Vector2(_facingDirection.X *
  _angle = (float)Math.Atan2(_facingDirection.Y, _faci
```

ACT I: Bliss

```
 if (_cameraPosition.Y - _playerPosition.Y > NoFlexZo
   y2 += _moveDirection.Y;

 if (_cameraPosition.Y - _playerPosition.Y < -NoFlexZ
   y2 += _moveDirection.Y;
 _cameraPosition = new Point(x2, y2);

foreach (var enemy in _enemies)
 {
 if (enemy.IsDoingNothing)
 {
   enemy.TicksUntilDoneDoingNothing--;
   if (enemy.TicksUntilDoneDoingNothing == 0)
   {
   enemy.IsDoingNothing = false;
   enemy.IsMoving = true;
   enemy.TicksUntilDoneMoving = 240;
```

ACT I: Bliss

```
} else if (enemy.IsMoving)
    {
      enemy.TicksUntilDoneMoving--;
      enemy.Position = enemy.Position + enemy.Directio

      if (enemy.TicksUntilDoneMoving == 0)
      {
        enemy.IsMoving = false;
        enemy.IsTurning = true;
        enemy.TicksUntilDoneTurning = 90;
      }
    } else if (enemy.IsTurning)
    {
      enemy.TicksUntilDoneTurning--;
      enemy.Direction = enemy.Direction.Rotate(1);
      if (enemy.TicksUntilDoneTurning == 0)
      {
```

ACT I: Bliss

```
_bullets.ForEach(p => { p.Position = new Vector2(p.Pos

var bulletsToDelete = _bullets.Where(x1 => Math.Abs(x1
  .ToArray();
foreach (var bulletToDelte in bulletsToDelete)
  _bullets.Remove(bulletToDelte);

if (_firing)
{
  var xDelta = _facingDirection.X*10f;
  var yDelta = _facingDirection.Y*10f;
  foreach (var gunAngle in _gunAngles)
  {
    var angle = (int)Math.Sqrt(_random.Next(0, 2*2*gun
    var direction = new Vector2(xDelta, yDelta).Rotate

    var bullet = new BulletStruct()
```

ACT I: Bliss

```
var bullet = new BulletStruct()
      {
        Position = new Vector2(_playerPosition.X + 16 * _facingDirection.X, _playerPosition.Y + 16 * _facingDirection.Y),
        Direction = direction
      };

      _bullets.Add(bullet);
    }
  }

  foreach (var bullet1 in _bullets.ToArray())
  {
    foreach (var enemy1 in _enemies)
    {
      Vector2 position1 = bullet1.Position;
      Vector2 position2 = enemy1.Position;
      if (((position1.X + 2 > position2.X - 16 && position1.X + 2 < position2.X + 16)
          || (position1.X - 2 < position2.X + 16 && position1.X - 2 > position2.X - 16)) &&
        ((position1.Y + 2 > position2.Y - 16 && position1.Y + 2 < position2.Y + 16)
          || (position1.Y - 2 < position2.Y + 16 && position1.Y - 2 > position2.Y - 16)))
      {
        _bullets.Remove(bullet1);
        enemy1.Health--;
        _collisionSplashes.Add(new CollisionSplashStruct()
        {
          Position = bullet1.Position,
          Direction = new Vector2() - bullet1.Direction,
          SplashCounter = 0
        });
      }
    }
  }

  foreach (var enemy2 in _enemies.ToArray())
  {
    if (enemy2.Health <= 0)
    {
      _enemies.Remove(enemy2);
      var explosionStruct = new ExplosionStruct(){ Position = enemy2.Position, Ticks = 0 };
      explosionStruct.Fragments = new List<Vector2>();
      for (int i = 0; i < 36; i++)
      {
        explosionStruct.Fragments.Add(new Vector2(1, 0).Rotate(_random.Next(0, 360)) * _random.Next(0, 10));
      }
      _explosions.Add(explosionStruct);
      _playerXp++;
    }
  }

  foreach (var splash in _collisionSplashes.ToArray())
  {
    splash.SplashCounter++;
```

ACT I: Bliss

```
  foreach (var splash in _collisionSplashes.ToArray())
  {
    splash.SplashCounter++;
    if (splash.SplashCounter > 10)
      _collisionSplashes.Remove(splash);
  }

  if (_triggerPowerUpText)
  {
    _powerUpCounter--;
    if (_powerUpCounter <= 0)
    {
      _triggerPowerUpText = false;
    }
  }
  if ((_playerLevel * _playerLevel + 1) / 3 < _playerXp)
  {
    _playerLevel++;
    _gunAngles.Add((int)Math.Sqrt(_random.Next(2, 250)));
    _triggerPowerUpText = true;
    _powerUpCounter = 90;
  }

  foreach (var explosion in _explosions.ToArray())
  {
    explosion.Ticks++;
    if (explosion.Ticks > 120)
    {
      _explosions.Remove(explosion);
    }
  }

  base.Update(gameTime);
}
```

ACT I: Bliss

In review:

- ✓ Easy to trace execution (just read from top-to-bottom)
- ✓ Easy to decide where to put the code

- ϴ Duplication, which causes bugs
- ϴ Duplication also makes deep restructuring difficult
- ϴ Classic spaghetti code

Aside: **When is it okay** to write “blissfully ignorant” code?

Aside: **What was the last bug you found** that was caused by duplicated code?

Questions?

Questions about how the application works?

Act 2: **Procedural Programming**

Code is grouped into **procedures** until there is no duplication. Group cohesive logic

~ ~ Lightning-fast Monogame tutorial ~ ~

Update()

Draw()

ACT II: Procedural Programming

```
protected override void Update(GameTime gameTime)
{
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
  Exit();

  var keyboardState = Keyboard.GetState();
  _facingDirection = new Vector2(0f, 0f);

  _moveDirection = new Point();
  if (keyboardState.IsKeyDown(Keys.Up))
  {
  _moveDirection.Y--;
  }
  if (keyboardState.IsKeyDown(Keys.Down))
  {
  _moveDirection.Y++;
  }
  if (keyboardState.IsKeyDown(Keys.Left))
  {
  _moveDirection.X--;
  }
  if (keyboardState.IsKeyDown(Keys.Right))
  {
  _moveDirection.X++;
  }

  if (keyboardState.IsKeyDown(Keys.W))
  {
  _moveDirection.Y--;
  }
  if (keyboardState.IsKeyDown(Keys.A))
  {
   _moveDirection.X--;
  }
  if (keyboardState.IsKeyDown(Keys.S))
  {
  _moveDirection.Y++;
  }
  if (keyboardState.IsKeyDown(Keys.D))
  {
  _moveDirection.X++;
  }

  _playerPosition = _playerPosition + _moveDirection;

  var mouseState = Mouse.GetState();
  _firing = mouseState.LeftButton == ButtonState.Pressed;

  var x = Math.Max(Math.Min(mouseState.Position.X, ScreenWidth), -ScreenWidth);
  var y = Math.Max(Math.Min(mouseState.Position.Y, ScreenHeight), -ScreenHeight);

  _facingDirection = new Vector2(0f, 0f);
  int xPositionOnScreen = (WidthMidpoint + ( _playerPosition.X -  _cameraPosition.X));
  int yPositionOnScreen = (HeightMidpoint + (_playerPosition.Y - _cameraPosition.Y));
  _facingDirection.X = ((float)(x - xPositionOnScreen) );
  _facingDirection.Y = ((float)(y - yPositionOnScreen) );
  float div = 1f/(float)Math.Sqrt(_facingDirection.X* facingDirection.X + _facingDirectio
   facingDirection = new Vector2( facingDirection.X * div,  facingDirection.Y * div);
  _angle = (float)Math.Atan2(_facingDirection.Y, _facingDirection.X);

  int x2 = _cameraPosition.X;
  int y2 = _cameraPosition.Y;
  if (_cameraPosition.X - _playerPosition.X > NoFlexZone)
  {
  x2 += _moveDirection.X;
  }

  if (_cameraPosition.X - _playerPosition.X < -NoFlexZone)
  {
  x2 += _moveDirection.X;
  }

  if (_cameraPosition.Y - _playerPosition.Y > NoFlexZone)
  {
  y2 += _moveDirection.Y;
  }

  if (_cameraPosition.Y - _playerPosition.Y < -NoFlexZone)
  {
  y2 += _moveDirection.Y;
  }
  _cameraPosition = new Point(x2, y2);

  foreach (var enemy in _enemies)
  {
  if (enemy.IsDoingNothing)
  {
    enemy.TicksUntilDoneDoingNothing--;
    if (enemy.TicksUntilDoneDoingNothing == 0)
    {
    enemy.IsDoingNothing = false;
    enemy.IsMoving = true;
    enemy.TicksUntilDoneMoving = 240;
    }
  } else if (enemy.IsMoving)
    {
      enemy.TicksUntilDoneMoving--;
      enemy.Position = enemy.Position + enemy.Direction;

      if (enemy.TicksUntilDoneMoving == 0)
      {
        enemy.IsMoving = false;
        enemy.IsTurning = true;
        enemy.TicksUntilDoneTurning = 90;
      }
    } else if (enemy.IsTurning)
    {
      enemy.TicksUntilDoneTurning--;
      enemy.Direction = enemy.Direction.Rotate(1);
      if (enemy.TicksUntilDoneTurning == 0)
      {
        enemy.IsTurning = false;
        enemy.IsDoingNothing = true;
        enemy.TicksUntilDoneDoingNothing = 60;
      }
    }
  }

  _bullets.ForEach(p => { p.Position = new Vector2(p.Position.X + p.Direction.X, p.Position.Y + p.Direction.Y); });

  var bulletsToDelete = _bullets.Where(x1 => Math.Abs(x1.Position.X) > GameBorder || Math.Abs(x1.Position.Y) > GameBorder)
    .ToArray();
  foreach (var bulletToDelte in bulletsToDelete)
    _bullets.Remove(bulletToDelte);

  if (_firing)
  {
    var xDelta =  facingDirection.X*10f;
    var yDelta = _facingDirection.Y*10f;
    foreach (var gunAngle in _gunAngles)
    {
      var angle = (int)Math.Sqrt(_random.Next(0, 2*2*gunAngle*gunAngle)) - gunAngle;
      var direction = new Vector2(xDelta, yDelta).Rotate(angle);

      var bullet = new BulletStruct()
      {
        Position = new Vector2(_playerPosition.X + 16 * _facingDirection.X, _playerPosition.Y + 16 * _facingDirection.Y),
        Direction = direction
```

```
        Position = new Vector2(_playerPosition.X + 16 * _facingDirection.X, _playe
        Direction = direction
      };

      _bullets.Add(bullet);
    }
  }

  foreach (var bullet1 in _bullets.ToArray())
  {
    foreach (var enemy1 in _enemies)
    {
      Vector2 position1 = bullet1.Position;
      Vector2 position2 = enemy1.Position;
      if (((position1.X + 2 > position2.X - 16 && position1.X + 2 < position2.X +
          || (position1.X - 2 < position2.X + 16 && position1.X - 2 > position2.X
        ((position1.Y + 2 > position2.Y - 16 && position1.Y + 2 < position2.Y + 16
          || (position1.Y - 2 < position2.Y + 16 && position1.Y - 2 > position2.Y
      {
        _bullets.Remove(bullet1);
        enemy1.Health--;
        collisionSplashes.Add(new CollisionSplashStruct()
        {
          Position = bullet1.Position,
          Direction = new Vector2() - bullet1.Direction,
          SplashCounter = 0
        });
      }
    }
  }

  foreach (var enemy2 in _enemies.ToArray())
  {
    if (enemy2.Health <= 0)
    {
      _enemies.Remove(enemy2);
      var explosionStruct = new ExplosionStruct(){ Position = enemy2.Position, Tic
      explosionStruct.Fragments = new List<Vector2>();
      for (int i = 0; i < 36; i++)
      {
        explosionStruct.Fragments.Add(new Vector2(1, 0).Rotate(_random.Next(0, 360
      }
      _explosions.Add(explosionStruct);
      _playerXp++;
    }
  }

  foreach (var splash in _collisionSplashes.ToArray())
  {
    splash.SplashCounter++;
    if (splash.SplashCounter > 10)
      _collisionSplashes.Remove(splash);
    splash.Direction.Y*=
  }

  if (_triggerPowerUpText)
  {
    _powerUpCounter--;
    if (_powerUpCounter <= 0)
    {
      _triggerPowerUpText = false;
    }
  }
  if ((_playerLevel * _playerLevel + 1) / 3 < _playerXp)
  {
    _playerLevel++;
    _gunAngles.Add((int)Math.Sqrt(_random.Next(2, 250)));
    _triggerPowerUpText = true;
    _powerUpCounter = 90;
  }

  foreach (var explosion in _explosions.ToArray())
  {
    explosion.Ticks++;
    if (explosion.Ticks > 120)
    {
      _explosions.Remove(explosion);
    }
  }

  base.Update(gameTime);
}
```

```
void Update(GameTime gameTime) {
  var k = ProcessKeybInput();
  var m = ProcessMouseInput();

  ApplyInputToPlayer(k, m);

  MovePlayer();
  MoveCamera();
  UpdateEnemies();
  UpdateBullets();
  DetectCollisions();
  KillEnemies();
  UpdateSplashes();
  CheckLevel();
  UpdateExplosions();
  base.Update(gameTime);
}
```

ACT II: Procedural Programming

217!

```
protected override void Update(GameTime gameTime)
{
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
  Exit();

  var keyboardState = Keyboard.GetState();
  _facingDirection = new Vector2(0f, 0f);

  _moveDirection = new Point();
  if (keyboardState.IsKeyDown(Keys.Up))
  {
  _moveDirection.Y--;
  }
  if (keyboardState.IsKeyDown(Keys.Down))
  {
  _moveDirection.Y++;
  }
  if (keyboardState.IsKeyDown(Keys.Left))
  {
  _moveDirection.X--;
  }
  if (keyboardS
  {
  _moveDirection
  }

  if (keyboardSta

  if
  {
  moveD

  }

  if
  {
  _moveDirecti

  if (_cameraPosi  layerP
  {
  y2 += _moveDi  .Y;
  }
  _cameraPosition = new Point(x2,

  foreach (var enemy in _enemies
  {
  if (enemy.IsDoingNothing)
  {
    enemy.TicksUntilDoneDoingNothing--;
    if (enemy.TicksUntilDoneDoingNothing == 0)
    {
    enemy.IsDoingNothing = false;
    enemy.IsMoving = true;
    enemy.TicksUntilDoneMoving = 240;
    }
  } else if (enemy.IsMoving)
    {
      enemy.TicksUntilDoneMoving--;
      enemy.Position = enemy.Position + enemy.Direction;

      if (enemy.TicksUntilDoneMoving == 0)
      {
        enemy.IsMoving = false;
        enemy.IsTurning = true;
        enemy.TicksUntilDoneTurning = 90;
      }
    } else if (enemy.IsTurning)
    {
      enemy.TicksUntilDoneTurning--;
      enemy.Direction = enemy.Direction.Rotate(1);
      if (enemy.TicksUntilDoneTurning == 0)
      {
        enemy.IsTurning = false;
        enemy.IsDoingNothing = true;
        enemy.TicksUntilDoneDoingNothing = 60;
      }
    }
  }

  _bullets.ForEach(p => { p.Position = new Vector2(p.Position.X + p.Direction.X, p.Position.Y + p.Direction.Y); });

  var bulletsToDelete = _bullets.Where(x1 => Math.Abs(x1.Position.X) > GameBorder || Math.Abs(x1.Position.Y) > GameBorder)
    .ToArray();
  foreach (var bulletToDelte in bulletsToDelete)
    _bullets.Remove(bulletToDelte);

  if (_firing)
  {
    var xDelta =  facingDirection.X*10f;
    var yDelta = _facingDirection.Y*10f;
    foreach (var gunAngle in _gunAngles)
    {
      var angle = (int)Math.Sqrt(_random.Next(0, 2*2*gunAngle*gunAngle)) - gunAngle;
      var direction = new Vector2(xDelta, yDelta).Rotate(angle);

      var bullet = new BulletStruct()
      {
        Position = new Vector2(_playerPosition.X + 16 * _facingDirection.X, _playerPosition.Y + 16 * _facingDirection.Y),
        Direction = direction
```

```
      {
        Position = new Vector2(_playerPosition.X + 16 * _facingDirection.X, _playe
        Direction = direction
      };

      _bullets.Add(bullet);
    }
  }

  foreach (var bullet1 in _bullets.ToArray())
  {
    foreach (var enemy1 in _enemies)
    {
      Vector2 position1 = bullet1.Position;
      Vector2 position2 = enemy1.Position;
      if (((position1.X > position2.X - 16 && position1.X + 2 < position2.X +
        || (position1.X - 2 < position2.X + 16 && position1.X - 2 > position2.X
      ((position1.Y > position2.Y - 16 && position1.Y + 2 < position2.Y + 16
        || (position1.Y - 2 < position2.Y + 16 && position1.Y - 2 > position2.Y

      _b                  bullet1);

                  hes                  ionSplashStruct()

                                                  bull

                                                                  sition, Tic

                                                                  indom.Next(0, 10));

                                                                                  250)));

                                                                                  s.ToArray

                                                                  > 120)

                                                                  (explosion);

                                                                  _Update(gameTime

                                                                  }
```

```
void Update(GameTime gameTime) {
  var k = ProcessKeybInput();
  var m = ProcessMouseInput();

  ApplyInputT(Play, k, m);

  MovePla
  MoveCam
  UpdateE
  UpdateBul
  DetectCollisions();
  KillEnemies();
  UpdateSplashes();
  CheckLevel();
  UpdateExplosions();
  base.Update(gameTime);
}
```

13

ACT II: Procedural Programming

```
_font = Content.Load<SpriteFont>("Font");
_texture = Content.Load<Texture2D>("a.png");
_enemyTexture = Content.Load<Texture2D>("b.png");

_bulletTexture = new Texture2D(GraphicsDevice, 4, 4);
_collisionSplashTexture = new Texture2D(GraphicsDevice, 3, 3);
_shrubberyTexture = Content.Load<Texture2D>("shrubbery.png");
var magenta = new Color(Color.Magenta, 1f);
var yellow = new Color(Color.Yellow, 1f);
var red = new Color(Color.Red, 1f);
_bulletTexture.SetData(new Color[16] { magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta, magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta });
_collisionSplashTexture.SetData(new Color[9] { red, red, red, red,
yellow, red, red, red, red });
_explosionTexture = new Texture2D(GraphicsDevice, 8, 8);
_explosionTexture.SetData(new Color[64] { red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red });
```

ACT II: Procedural Programming

```
_font = Content.Load<SpriteFont>("Font");
_texture = Content.Load<Texture2D>("a.png");
_enemyTexture = Content.Load<Texture2D>("b.png");

_bulletTexture = new Texture2D(GraphicsDevice, 4, 4);
_collisionSplashTexture = new Texture2D(GraphicsDevice, 3, 3);
_shrubberyTexture = Content.Load<Texture2D>("shrubbery.png");
var magenta = new Color(Color.Magenta, 1f);
var yellow = new Color(Color.Yellow, 1f);
var red = new Color(Color.Red, 1f);
_bulletTexture.SetData(new Color[16] { magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta, magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta });
_collisionSplashTexture.SetData(new Color[9] { red, red, red, red,
yellow, red, red, red, red });
_explosionTexture = new Texture2D(GraphicsDevice, 8, 8);
_explosionTexture.SetData(new Color[64] { red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red });
```

ACT II: Procedural Programming

```
_font = Content.Load<SpriteFont>("Font");
_texture = Content.Load<Texture2D>("a.png");
_enemyTexture = Content.Load<Texture2D>("b.png");
```

```
_bulletTexture = new Texture2D(GraphicsDevice, 4, 4);
_collisionSplashTexture = new Texture2D(GraphicsDevice, 3, 3);
_shrubberyTexture = Content.Load<Texture2D>("shrubbery.png");
var magenta = new Color(Color.Magenta, 1f);
var yellow = new Color(Color.Yellow, 1f);
var red = new Color(Color.Red, 1f);
_bulletTexture.SetData(new Color[16] { magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta, magenta, magenta, magenta,
magenta, magenta, magenta, magenta, magenta }); 
_collisionSplashTexture.SetData(new Color[9] { red, red, red, red,
yellow, red, red, red, red }); 
_explosionTexture = new Texture2D(GraphicsDevice, 8, 8);
_explosionTexture.SetData(new Color[64] { red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, red,
red, red });
```

```
private void LoadFont() {
  _font = LoadFontByName("Font");
}

private void LoadTexturesFromFile() {
  _texture = LoadTextureFromFile("a.png");
  _enemyTexture = LoadTextureFromFile("b.png");
  _shrubberyTexture = LoadTextureFromFile("shrubbery.png");
}

private void LoadTexturesFromArray() {
  _bulletTexture = CreateSquareTexture(Color.Magenta, BulletSize);
  _collisionSplashTexture = CreateSquareTexture(Color.Red, SplashSize);
  _explosionTexture = CreateSquareTexture(Color.Red, FragmentSize);
}
```

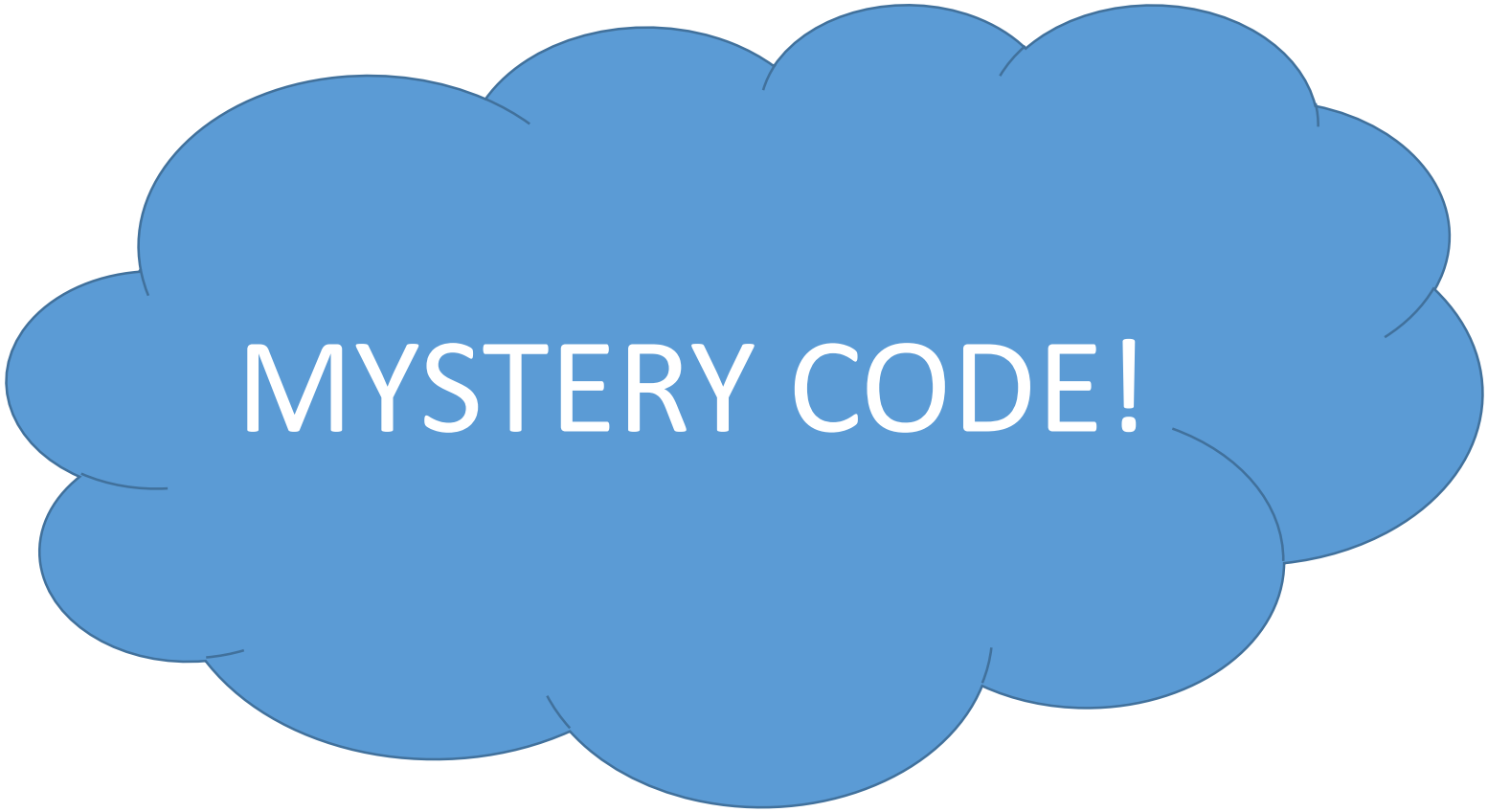
ACT II: Procedural Programming

```
_bulletTexture = new Texture2D(GraphicsDevice, 4, 4);
_collisionSplashTexture = new Texture2D(GraphicsDevice, 3, 3);
var magenta = new Color(Color.Magenta, 1f);
var yellow = new Color(Color.Yellow, 1f);
var red = new Color(Color.Red, 1f);
_bulletTexture.SetData(new Color[16] { magenta, magenta, magenta,
 _collisionSplashTexture.SetData(new Color[9] { red, red, red, red,
 _explosionTexture = new Texture2D(GraphicsDevice, 8, 8);
_explosionTexture.SetData(new Color[64] { red, red, red, red, red,
```

```
private Texture2D CreateSquareTexture(Color color, int size) {
  var texture = new Texture2D(GraphicsDevice, size, size);

  texture.SetData(
    Enumerable.Range(0, size * size)
    .Select(cell => color)
    .ToArray());

  return texture;
}
```

ACT II: Procedural Programming

```
y = random.Next(0, 2);
if (y == 0)
  y = -1;
```



ACT II: Procedural Programming

```
y = random.Next(0, 2);
if (y == 0)
  y = -1;
```

```
y = GenerateRandomNegativeOrPositiveOne(random);
```

ACT II: Procedural Programming

```
y = random.Next(0, 2);
if (y == 0)
  y = -1;
```

```
y = GenerateRandomNegativeOrPositiveOne(random);
```

```
int GenerateRandomNegativeOrPositiveOne(Random random) {
  return GetRandomBool(random) ? 1 : -1;
}
bool GetRandomBool(Random random) {
  return NextRandomNumber(random, 1) == 1;
}
int NextRandomNumber(Random random, int maxValue) {
  return NextRandomNumber(random, 0, maxValue);
}
int NextRandomNumber(Random random, int min, int max) {
  return random.Next(min, max + 1);
}
```

ACT II: Procedural Programming

```
y = random.Next(0, 2);
if (y == 0)
  y = -1;
```

```
y = GenerateRandomNegativeOrPositiveOne(random);
```

```
y = random.Next(0, 1 + 1) == 1 ? 1 : -1;
```

ACT II: Procedural Programming

```
y = random.Next(0, 1 + 1) == 1 ? 1 : -1;
```

```
y = GenerateRandomNegativeOrPositiveOne(random);
```

ACT II: Procedural Programming

In review:

- ✓ Easier to reason about code that is grouped by procedure (this means easier troubleshooting)
- ✓ Eliminates duplication, which means fewer bugs and fewer things to remember
- ✓ (limited) Encapsulation

- ϴ Harder to read from top-to-bottom
- ϴ Some friction moving the code for two reasons:
  - ϴ Mechanically difficult
  - ϴ Not sure where to put the abstracted code

Aside: **Eliminate duplication** by moving duplicated logic into methods. It's brainless (no analysis paralysis), and you know when you're done.

Aside: if we all agree duplication creates bugs, is easily identifiable, and easy to fix, why is there **so much duplication** in my codebase? Can we solve this problem?

Questions?

Act 3: Objects

It is **impossible** to describe object-oriented programming in C#.

Act 3: **Objects**

OOP as defined by interview answers:
inheritance, **polymorphism**,
information hiding, **encapsulation**, and
abstraction

Act 3: **Objects**

OOP as defined by father of OOP: objects are **independent actors sending messages** to each other

Act 3: **Objects**

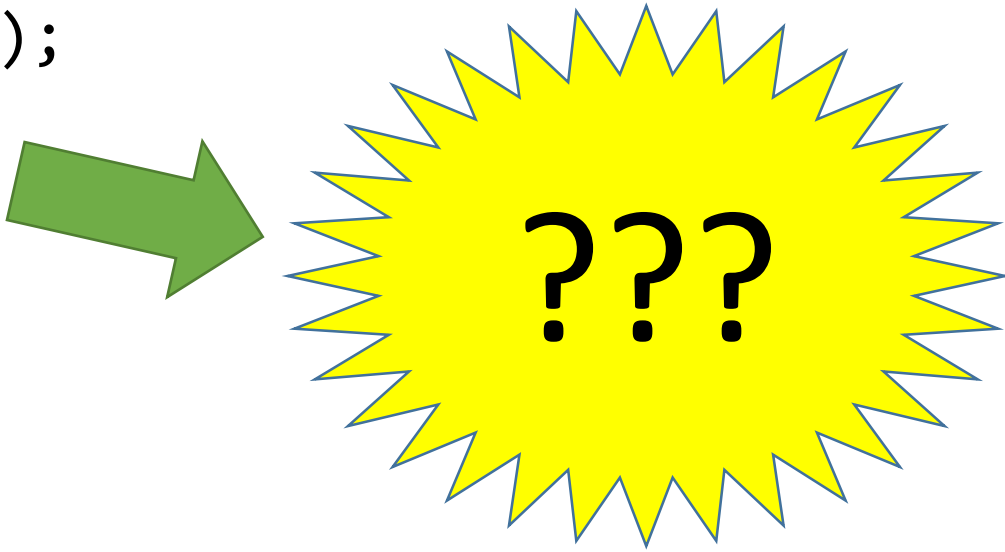
Dave West in Object Thinking describes objects as promoting **encapsulation** and **composability**

Act 3: Objects

Move cohesive methods into Helper classes. **Move methods with common state** into objects. If the methods use 3rd-party objects, **create services**.

ACT III: Objects

```
void Update(GameTime gameTime) {
  var k = ProcessKeybInput();
  var m = ProcessMouseInput();

  ApplyInputToPlayer(k, m);

  MovePlayer();
  MoveCamera();
  UpdateEnemies();
  UpdateBullets();
  DetectCollisions();
  KillEnemies();
  UpdateSplashes();
  CheckLevel();
  UpdateExplosions();
  base.Update(gameTime);
}
```



ACT III: Objects

```
public static class MathHelper {
  public static Vector2 ShrinkVectorTo1Magnitude(Vector2 vector) {
    var magnitude = 1f / (float)Math.Sqrt(vector.X * vector.X
                                          + vector.Y * vector.Y);
    return  vector * magnitude;
  }

  public static float ConvertToAngleInRadians(Vector2 direction) {
    return (float)Math.Atan2(direction.Y, direction.X);
  }

  public static Vector2 Rotate(Vector2 v, float degrees) {
    float Deg2Rad = ((float)(2 * Math.PI)/ 360f);
    float sin = (float)Math.Sin(degrees * Deg2Rad);
    float cos = (float)Math.Cos(degrees * Deg2Rad);

    float tx = v.X;
    float ty = v.Y;
    v.X = (cos * tx) - (sin * ty);
    v.Y = (sin * tx) + (cos * ty);
    return v;
  }
```

ACT III: Objects

```
public static class MathHelper {
  public static Vector2 ShrinkVectorTo1Magnitude(Vector2 vector)
  public static float ConvertToAngleInRadians(Vector2 direction)
  public static Vector2 Rotate(Vector2 vector, float degrees)
}
```

ACT III: Objects

```
public class RandomNumberService : IRandomNumberService {
    private readonly Random _random;
    public RandomNumberService() {
        _random = new Random();
    }
    public RandomNumberService(int seed) {
        _random = new Random(seed);`
    }

    public int NextRandomNumberBetweenPositiveAndNegative(int value) {
        return NextRandomNumber(value);
    }
    public bool GetRandomBool() {
        return NextRandomNumber(1) == 1;
    }
    public double GenerateRandomNumberClusteredTowardZero(int max) {
        return Math.Sqrt(NextRandomNumber(max * max));
    }

    public int NextRandomNumber(int minValue, int maxValue) {
        return _random.Next(minValue, maxValue + 1);
    }
}
```

FUN FACT:
there is a bug
in this code!

ACT III: Objects

```
public class RandomNumberService : IRandomNumberService {
  public RandomNumberService()
  public RandomNumberService(int seed)
  public int NextRandomNumberBetweenPositiveAndNegative(int value)
  public bool GetRandomBool()
  public double GenerateRandomNumberClusteredTowardZero(int max)
  public int NextRandomNumber(int minValue, int maxValue)
}
```

ACT III: Objects

```
public class Bullet {
  public Bullet(Vector2 position, Vector2 direction) {
    Position = position;
    Direction = direction;
  }

  public Vector2 Position { get; private set; }
  public Vector2 Direction { get; private set; }

  public void Move() {
    Position = Position + Direction;
  }
}
```

ACT III: Objects

```
public class Bullet {
  public Bullet(Vector2 position, Vector2 direction)

  public Vector2 Position { get; }
  public Vector2 Direction { get; }

  public void Move()
}
```

ACT III: Objects

```
public class DrawService : IDrawService {
  public DrawService(SpriteBatch spriteBatch, GraphicsDevice graphicsDev
    _spriteBatch = spriteBatch; _graphicsDevice = graphicsDevice;
  }

  public void DrawEntityWithRotation(Texture2D texture, Vector2 position
    _spriteBatch.Draw(texture, position, new Rectangle(0, 0, playerSize,
      new Color(Color.White, 1f), MathHelper.ConvertToAngleInRadians(dir
      new Vector2(playerSize/2, playerSize/2), 1.0f, SpriteEffects.None,
  }

  public void InitializeFrame(Point cameraPosition, int widthMidpoint, i
    _graphicsDevice.Clear(backgroundColor);

    //http://www.david-amador.com/2009/10/xna-camera-2d-with-zoom-and-ro
    var transform = Matrix.CreateTranslation(new Vector3(-cameraPosition
                    Matrix.CreateRotationZ(0)*
                    Matrix.CreateScale(new Vector3(1, 1, 1))*
                    Matrix.CreateTranslation(new Vector3(widthMidpoint,
    _spriteBatch.Begin(SpriteSortMode.Deferred, null, null, null, null,
  }
}
```

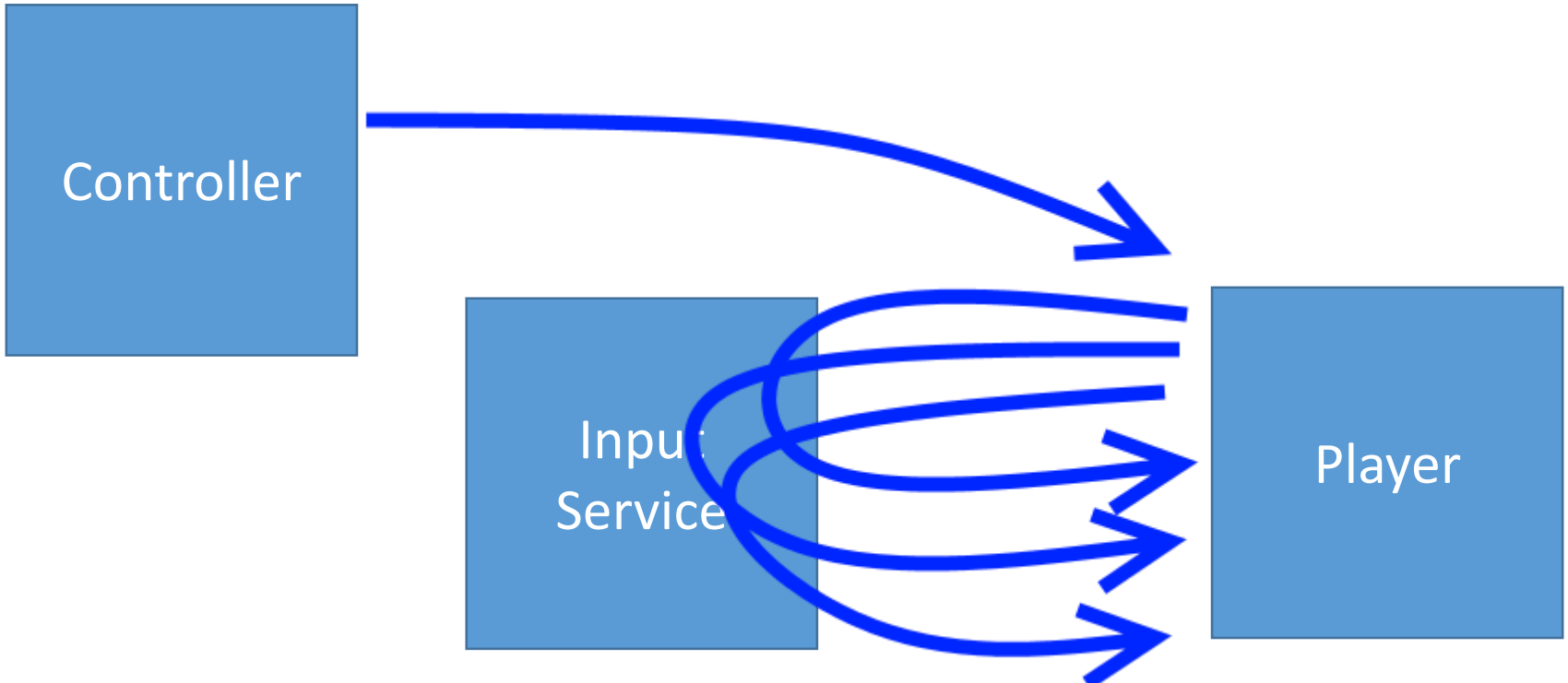
ACT III: Objects

```
ice) {


, Vector2 direction, int playerSize) {
 playerSize),
ection),
 1);


 heightMidpoint, Color backgroundColor) {


tation/
.X, -cameraPosition.Y, 0))*


heightMidpoint, 0));
null, transform);
```

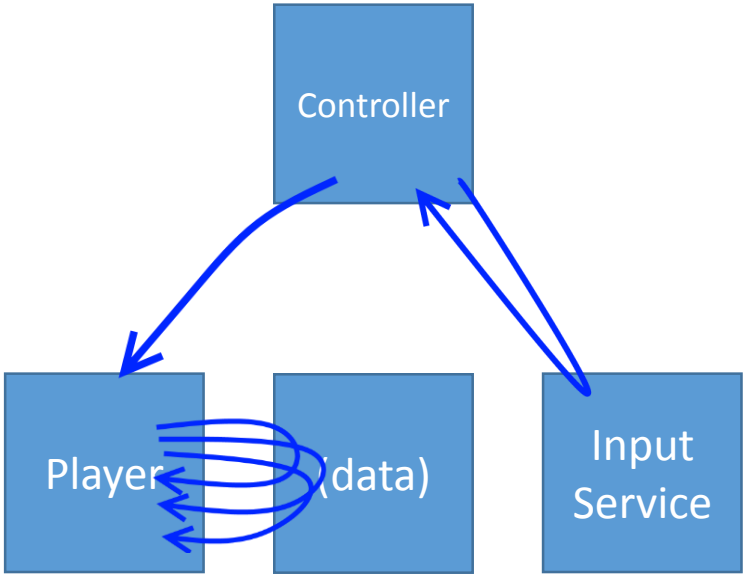
ACT III: Objects

```
public class DrawService : IDrawService {
  void DrawEntityWithRotation(Texture2D texture,
                              Vector2 position,
                              Vector2 direction,
                              int size)

  void InitializeFrame(Point cameraPosition,
                       int widthMidpoint,
                       int heightMidpoint,
                       Color backgroundColor)
}
```

```
protected override void Update(GameTime gameTime) {
  _player.Update(_inputService);
}


//Player.cs
public void Update(InputService inputService) {
  MoveDirection = inputService.GetMoveDirection();
  IsFiring = inputService.IsFiring();
  FacingDirection = inputService.GetMouseFacingDirection();
}
```



```
//in GameController.cs
void Update(GameTime gameTime) {
  var input = _inputService
        .ProcessInput(Midpoint,
                      _player.Position,
                      _camera.Position);

  _player.Update(input);
}

//in Player.cs
void Update(InputStruct input) {
  MoveDirection = input.MoveDirection;
  IsFiring = input.IsFiring;
  FacingDirection = input.PlayerFacingDirection;
}
```




```
public class InputDto {
  public Point MoveDirection { get; set; }
  public bool IsFiring { get; set; }
  public Vector2 PlayerFacingDirection { get; set; }
}
```

ACT III: Objects

In review:

SCORECARD:

Green: ok
Yellow: caution
Red: abort

- ✓ Encapsulation – you can work with parts of the system without fully understanding how they work
- ✓ Composability – you can compose complex behavior from objects

- ϴ Object design is an art, and requires practice and study to become comfortable
- ϴ Wrongly-abstracted objects are worse than spaghetti
- ϴ OOP is often taught wrong

Aside: how do you know your design is correct?

How long after do you feel your object designs ‘settle’?

Aside: What do you use to help you design objects?

- CRC cards
- diagrams
- software products
- gut

Aside: do the SOLID principles help **you** design better objects?

Aside: how did you learn to program with objects?

Aside: do you use **extension methods** (instead of old-fashioned static methods)? What are the problems you encounter when using them?

Questions?

Every question welcome, **except from functional programmers**

Act 4: DDD

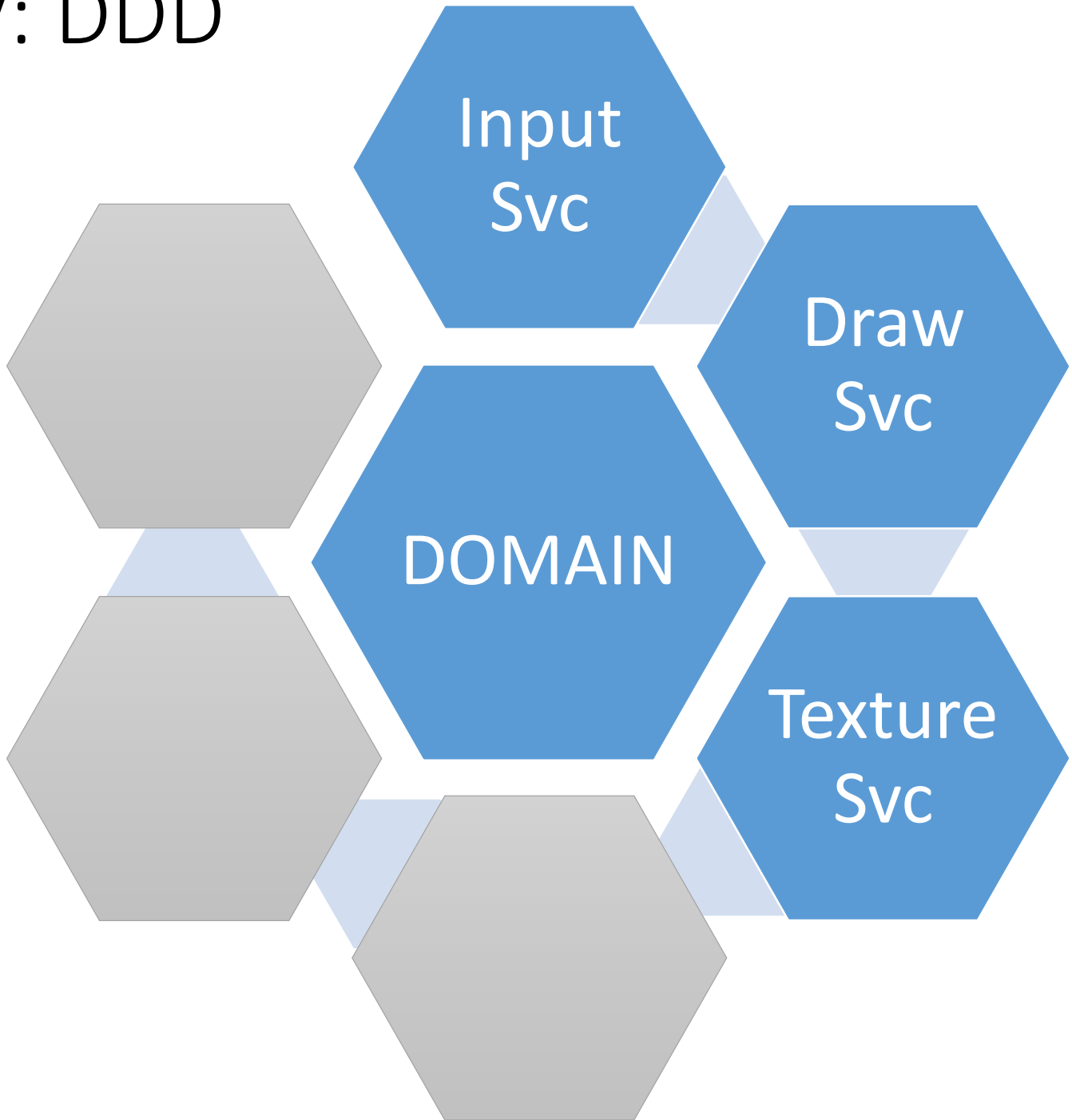
“ (DDD) is a collection of principles and patterns that help developers craft elegant object systems.”

Quote from @laribee - https://msdn.microsoft.com/en-us/magazine/dd419654.aspx

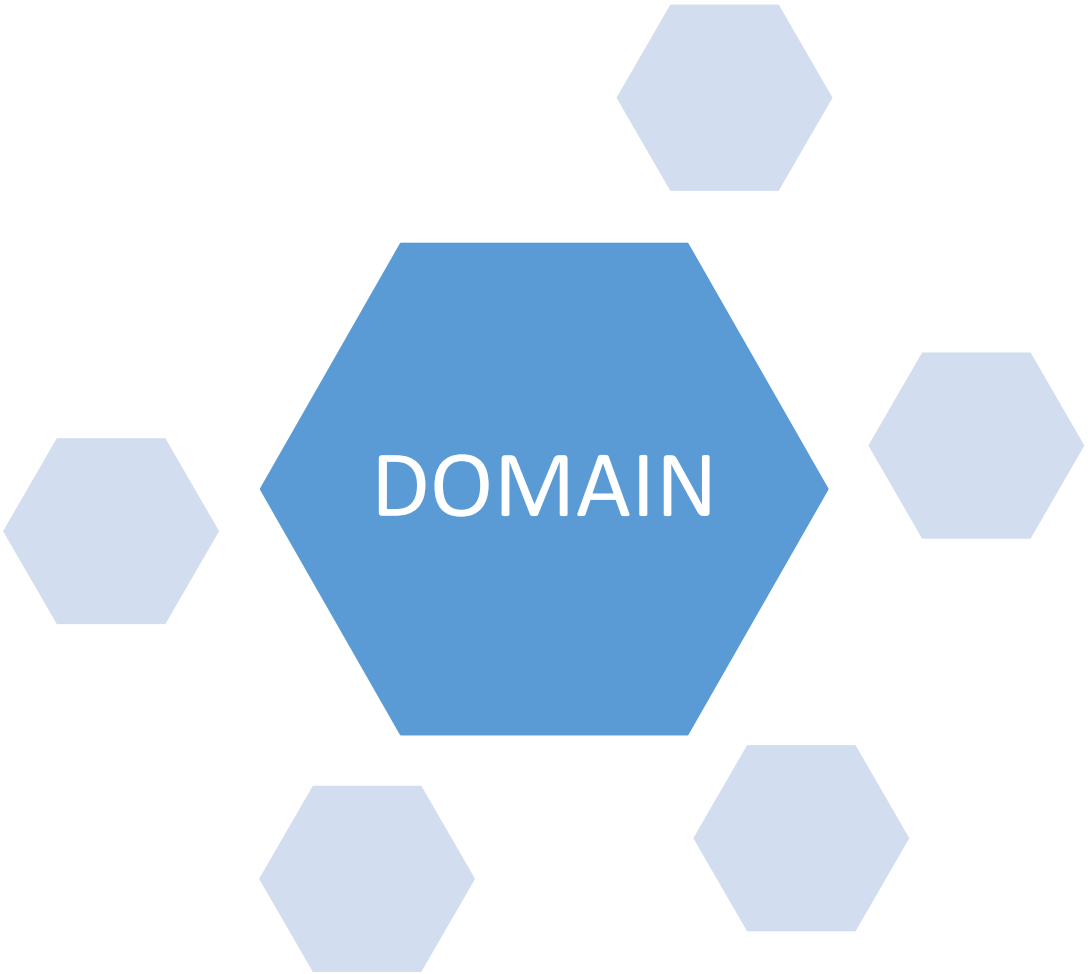
Act 4: **DDD**

Read the entire DDD book by Evans*, plus a thousand blog posts, then **apply DDD principles to codebase**.

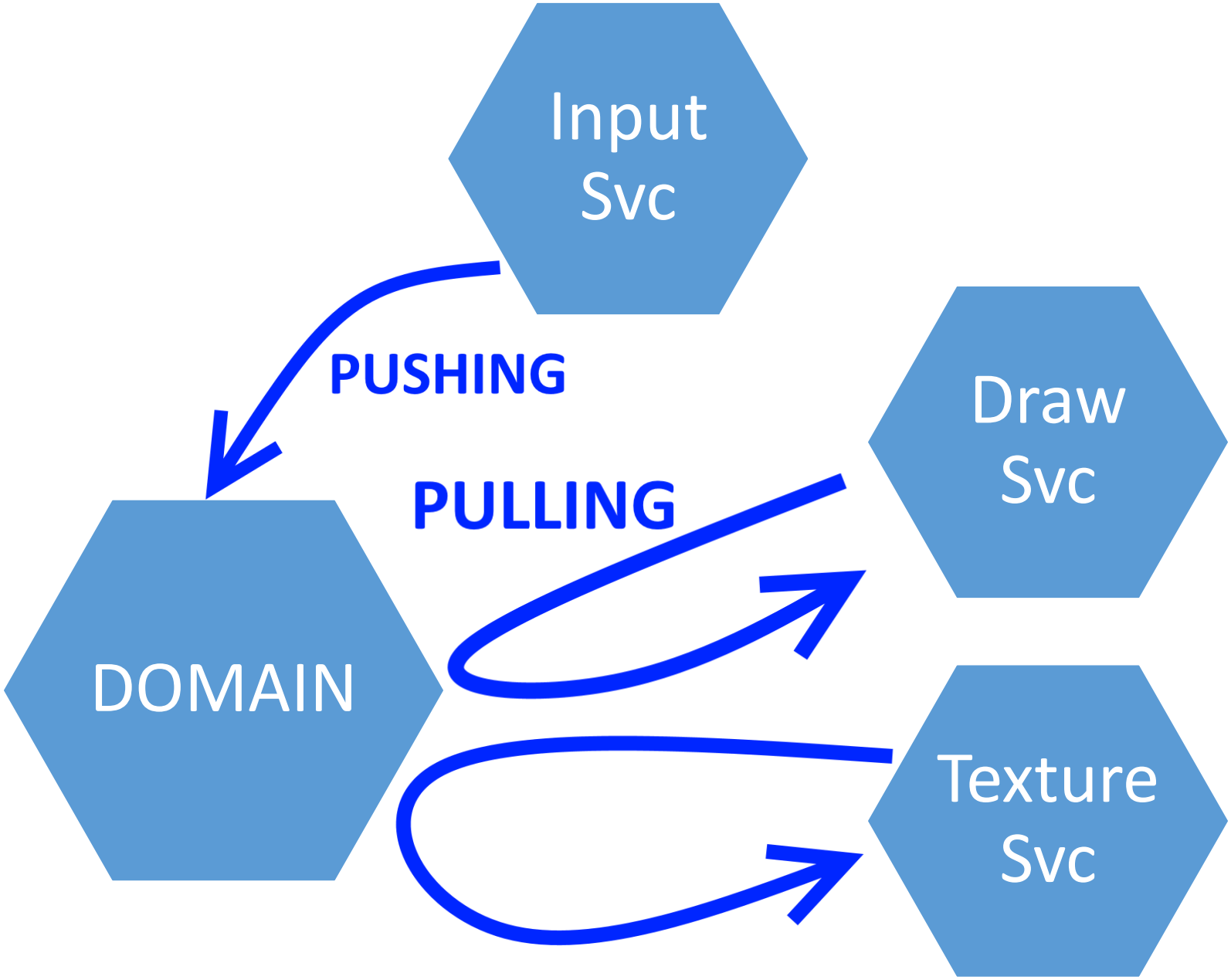
ACT IV: DDD



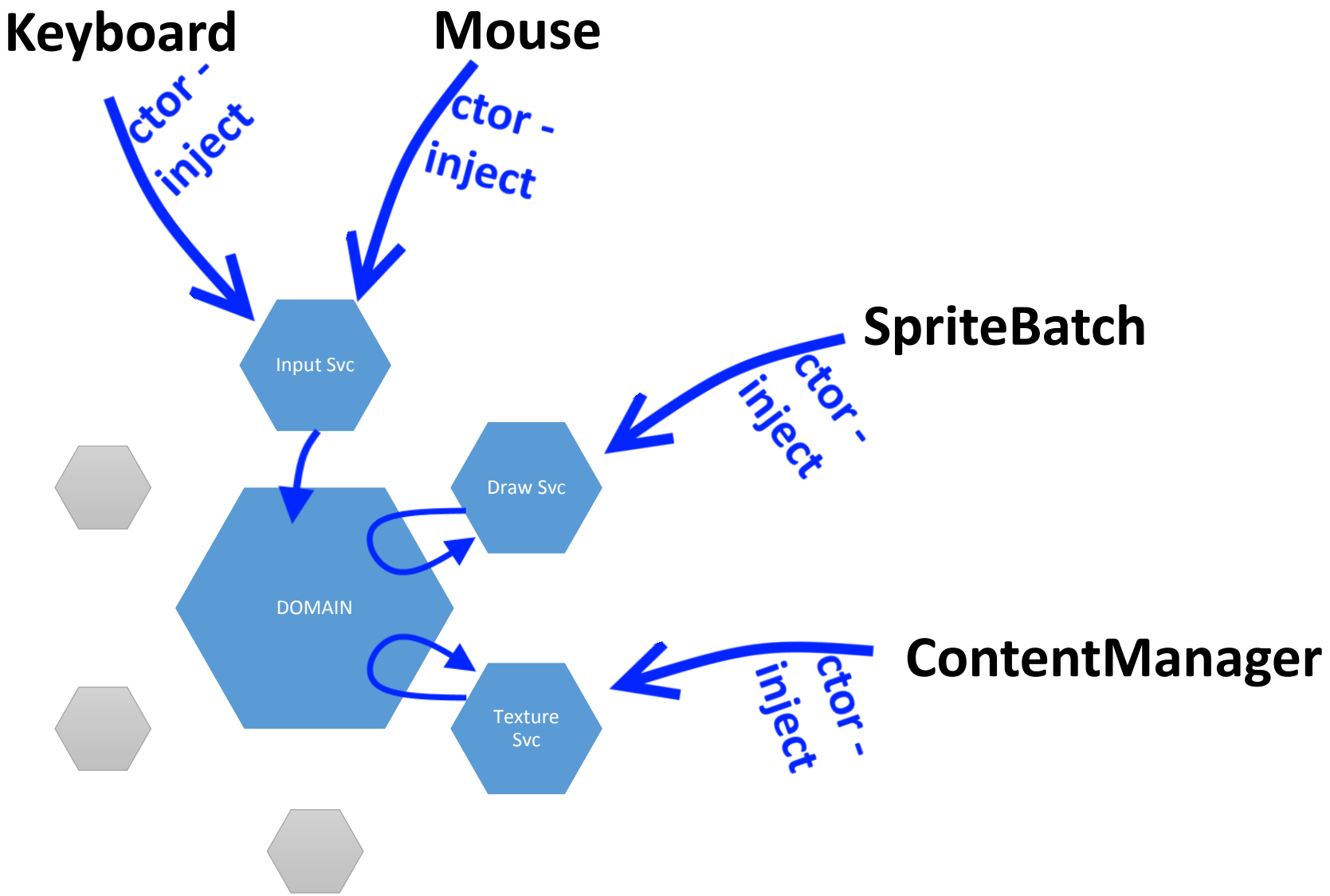
ACT IV: DDD



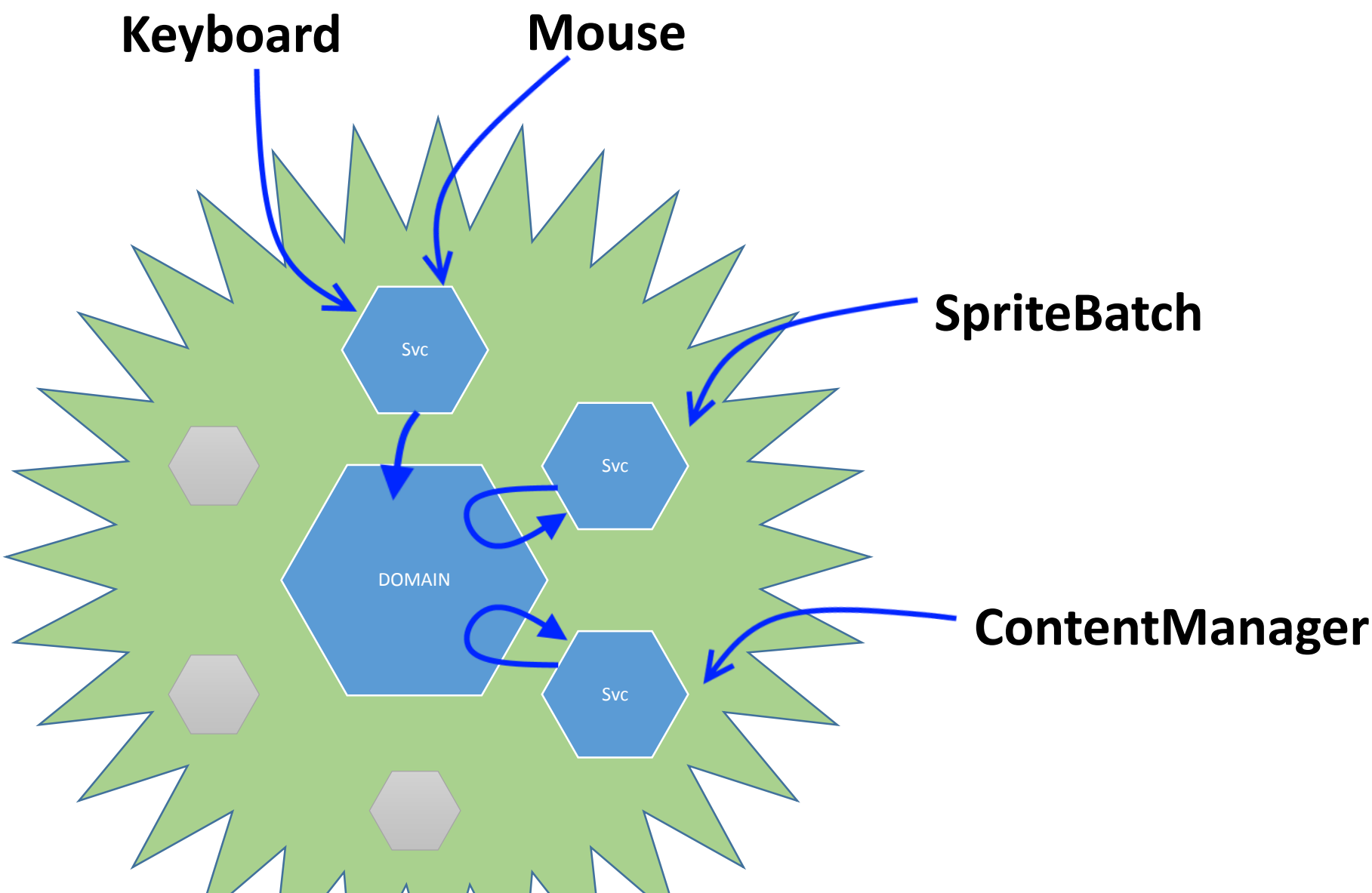
ACT IV: DDD



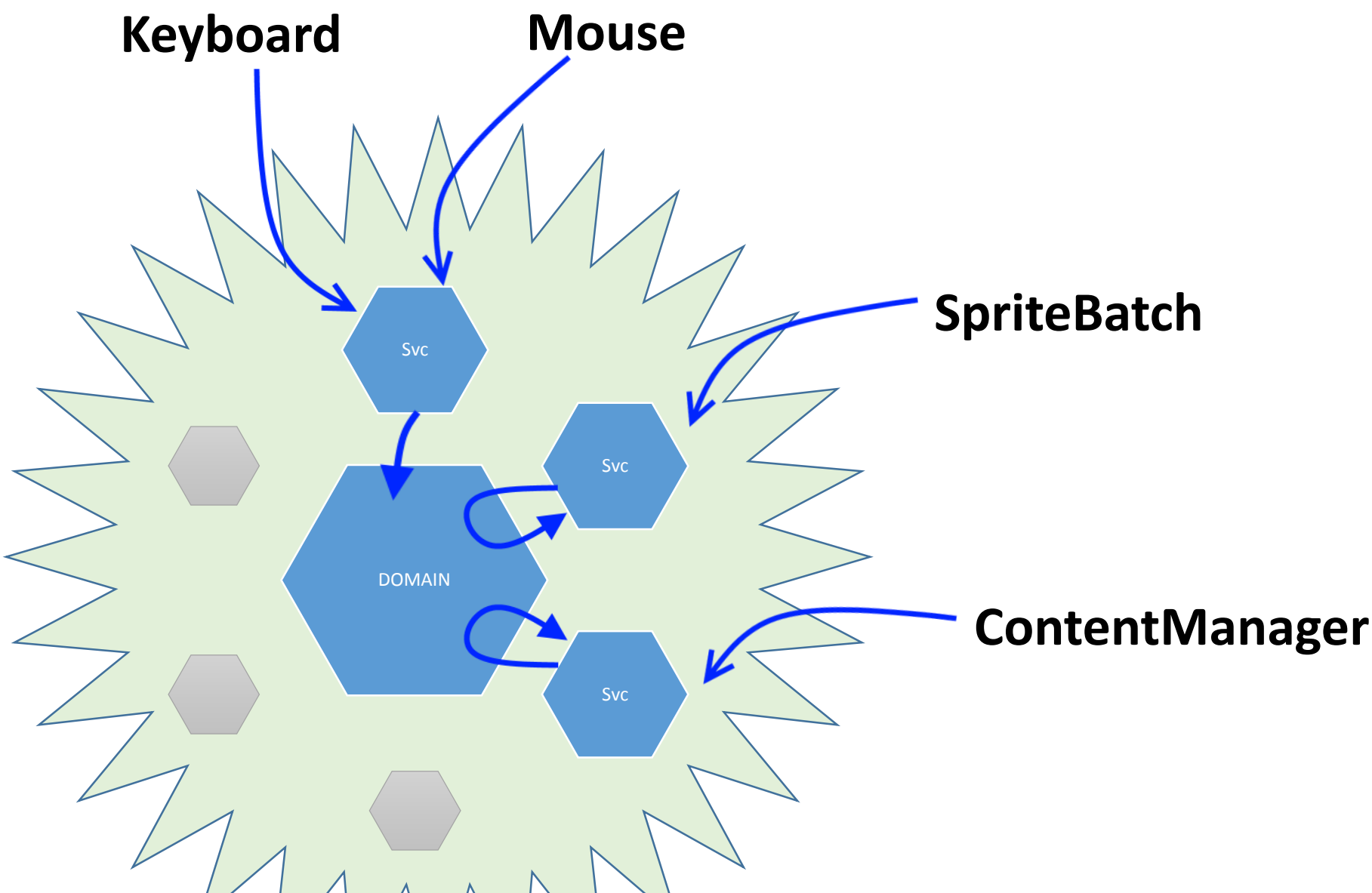
ACT IV: DDD


Keyboard
Mouse
ctor - inject
ctor - inject
Input Svc
SpriteBatch
ctor - inject
Draw Svc
DOMAIN
ContentManager
ctor - inject
Texture Svc

ACT IV: DDD


Keyboard
Mouse
SpriteBatch
ContentManager
Svc
Svc
Svc
DOMAIN

ACT IV: DDD

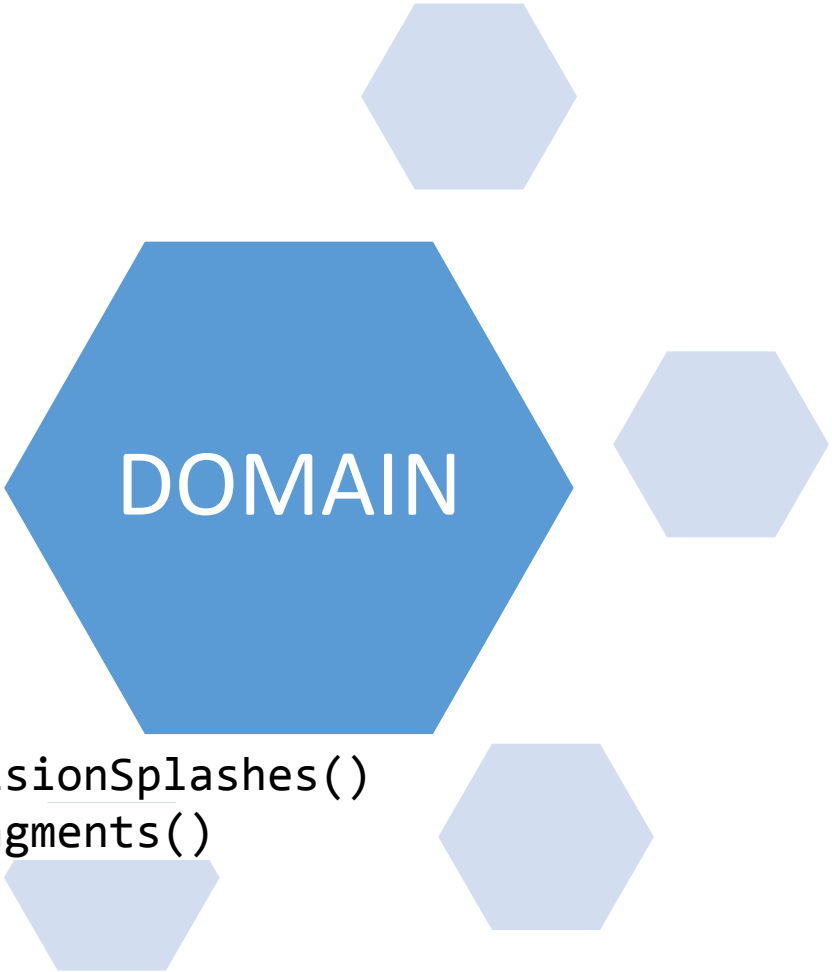

Keyboard
Mouse
SpriteBatch
ContentManager
Svc
Svc
Svc
DOMAIN

Act 4: **DDD**

Create a domain that references nothing outside itself. **Create services** that adapt third-party libraries for use with our domain. **Glue** services and domain to everything else.

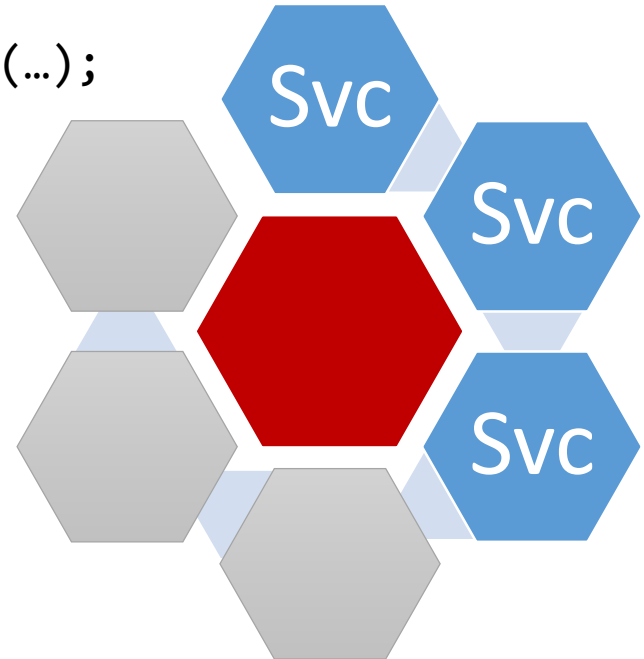
ACT IV: DDD

```
public class Game {

  //called from Update()
  void Update(InputDto input)

  //called from Draw()
  Point GetCameraPosition()
  Point GetPlayerPosition()
  Vector2 GetPlayerFacingDirection()
  IEnumerable<Bullet> GetBullets()
  IEnumerable<Enemy> GetEnemies()
  IEnumerable<Shrubbery> GetShrubbery()
  IEnumerable<CollisionSplash> GetCollisionSplashes()
  IEnumerable<ExplosionFragment> GetFragments()
  bool ShouldTriggerPowerUpText()

  //object design is hard – not sure where to put this
  bool OutOfBounds(float position)
}
```



ACT IV: DDD

```
public class MonogameDemoGame : Game {

  protected override void Update(…) {
    var input = _inputService.ProcessInput();
    _game.Update(input);
  }

  protected override void Draw(…) {
    var vm = ViewModelMapper.CreateViewModel(_game);
    _drawService.InitializeFrame(vm.CameraPosition);
    foreach (var entity in vm.Entities)
      if (entity.HasRotation)
        _drawService.DrawEntityWithRotation(…);
      else
        _drawService.DrawEntity(…);
  }
}
```



ACT IV: DDD

In review:

SCORECARD:

Green: ok
Yellow: caution
Red: abort

✓ DDD provides better guiding principles than "naked" OO, which means your abstractions are better, which means you can think in the abstract "ubiquitous language", which lets you solve problems you otherwise couldn't

ϴ Large learning curve, which means that in a large endeavor, your team will create many bad domain models

ϴ Bad domain models are a tragedy – you get none of the benefits, but mental overhead and N+1s

ϴ Overhead

Aside: Is F# the “pit of success” we need?

If yes, why aren’t we using it? How has F# changed you?

If ambivalent, why do you think anyone bothers with F#?

Questions?

Questions about how the application works?

Act 5: **DSLs**

Domain-specific language

A **specialized language** designed to match your solution space.

ACT V: Domain-specific languages

```
Entity Player
  color < Cyan
  > Mouse1
    fire
  > Keyboard.W
    move -1 0
  > Keyboard.A
    move 0 -1
  > Keyboard.S
    move 1 0
  > Keyboard.D
    move 0 1

  > level_up
    #todo implement
```

ACT V: Domain-specific languages

```
DefineEntity("Player")
                .Color(@cyan)
                .On(@mouse1, () => Fire())
                .On(@w, () => Move(-1, 0))
                .On(@a, () => Move(0, -1))
                .On(@s, () => Move(1, 0))
                .On(@d, () => Move(0, 1))
                .On(@level_up, () =>
                {
                    /* todo implement */
                });
```

ACT V: Domain-specific languages

Implementing an external DSL requires one of the following:

1. Irony
2. M Lang (Oslo)
3. JetBrains MPS
4. ANTLR
5. Sending your ASTs to Roslyn

Aside: if DSLs are so scary to implement, are they **ever** needed?

```
@model WebApplication1.Models.ChangePasswordViewModel
@{
    ViewBag.Title = "Change Password";
}

<h2>@ViewBag.Title.</h2>

@using (Html.BeginForm("ChangePassword", "Manage", FormMethod.Post, new
{
    @Html.AntiForgeryToken()
    <h4>Change Password Form</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.OldPassword, new { @class = "col-md-2 cont
        <div class="col-md-10">
            @Html.PasswordFor(m => m.OldPassword, new { @class = "form-c
        </div>
    </div>
}
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

```xml
<UserControl x:Class="MonogameDemoGame.DslInAction"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300">
    <Grid>

    </Grid>
</UserControl>
```

Aside: Workflow Foundation is a DSL (or at its core, a platform to help you build a DSL). **Does it suffer the same problems** as other DSLs?

Aside: what is the worst DSL you've ever used? Or just what is the most recent bad DSL that left its scars on you?

What is the best DSL you've seen?

ACT V: Domain-specific languages

In review:

✓ With the correct abstractions, DSLs elevate your thinking

ϴ With incorrect abstractions, DSLs are crippling

ϴ You are bad at making DSLs

Questions?

Epilogue:
Takeaways

Make war against the inner laziness within you

Epilogue: Takeaways

Remove duplication as your first priority–the rest follows.

Epilogue:
Takeaways

Make the smallest change possible

Epilogue:
Takeaways

Choose the simplest abstraction that works

Epilogue:

Takeaways

Don’t be ashamed to create Helper classes

Epilogue: Takeaways

Unspoken rule that must now be spoken:

You Ain’t Gonna Need It (YAGNI)

Epilogue:
Takeaways

Don’t **feel** rushed

Epilogue:
Takeaways

Use your unit test projects as **practice** working with abstractions

Epilogue:
Takeaways

Find Usages

Epilogue:
Takeaways

Improve your build & deployment process.

~ ~ Thank you ~ ~

github.com/pseale/presentation-architecture-madness

Full refunds available at the box office