Modular Code Design

Building Reusable, Maintainable Components

The foundation of scalable software architecture

Why Modular Design?

> **"Programs should be written for people to read, and only incidentally for machines to execute"** - Abelson & Sussman

- **Maintainability** - Easier to understand and modify
- **Reusability** - Components can be used in multiple contexts
- **Testability** - Isolated units are easier to test
- **Scalability** - Teams can work on different modules

The Problem

Monolithic Code Issues:

- 🍝 Spaghetti code with tight coupling
- 🔧 Changes in one area break unrelated features
- 😰 Fear of modifying existing code
- 👥 Difficult for teams to work in parallel
- 🐛 Hard to isolate and fix bugs

Result: Development slows down over time

The SOLID Principles

Foundation of Good Design

S - Single Responsibility Principle

O - Open/Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle

Single Responsibility Principle

One Reason to Change

> "A class should have only one reason to change"

Bad Example:

```
class User {
  constructor(name, email) { ... }
  save() { /* database logic */ }
  sendEmail() { /* email logic */ }
  validateEmail() { /* validation logic */ }
}
```

Good Example:

```
class User { constructor(name, email) { ... } }
class UserRepository { save(user) { ... } }
class EmailService { send(user, message) { ... } }
class EmailValidator { validate(email) { ... } }
```

Open/Closed Principle

Open for Extension, Closed for Modification

Bad Example:

```
class PaymentProcessor {
  process(payment) {
    if (payment.type === 'credit') {
      // credit card logic
    } else if (payment.type === 'paypal') {
      // paypal logic
    }
    // Adding new payment type requires modifying this class
  }
}
```

Open/Closed - Good Example

```
// Abstract base
class PaymentProcessor {
  process(payment) {
    throw new Error('Must implement process method');
  }
}

// Concrete implementations
class CreditCardProcessor extends PaymentProcessor {
  process(payment) { /* credit card logic */ }
}

class PayPalProcessor extends PaymentProcessor {
  process(payment) { /* paypal logic */ }
}

// Factory pattern
class PaymentFactory {
  static create(type) {
    const processors = {
      'credit': CreditCardProcessor,
      'paypal': PayPalProcessor
    };
    return new processors[type]();
  }
}
```

Liskov Substitution Principle

Subtypes Must Be Substitutable

Example:

```
class Rectangle {
  setWidth(width) { this.width = width; }
  setHeight(height) { this.height = height; }
  getArea() { return this.width * this.height; }
}

class Square extends Rectangle {
  setWidth(width) {
    this.width = width;
    this.height = width; // Maintains square property
  }
  setHeight(height) {
    this.width = height;
    this.height = height;
  }
}
```

Interface Segregation Principle

Don't Force Unnecessary Dependencies

Bad Example:

```
class AllInOneInterface {
  read() { ... }
  write() { ... }
  execute() { ... }
  compress() { ... }
}

// ReadOnlyFile forced to implement methods it doesn't need
class ReadOnlyFile implements AllInOneInterface {
  read() { /* implementation */ }
  write() { throw new Error('Not supported'); }
  execute() { throw new Error('Not supported'); }
  compress() { throw new Error('Not supported'); }
}
```

Interface Segregation - Good Example

```
// Specific interfaces
class Readable {
  read() { throw new Error('Must implement'); }
}

class Writable {
  write() { throw new Error('Must implement'); }
}

class Executable {
  execute() { throw new Error('Must implement'); }
}

// Classes implement only what they need
class ReadOnlyFile extends Readable {
  read() { /* implementation */ }
}

class ReadWriteFile extends Readable, Writable {
  read() { /* implementation */ }
  write() { /* implementation */ }
}
```

Dependency Inversion Principle

Depend on Abstractions, Not Concretions

Bad Example:

```
class OrderService {
  constructor() {
    this.emailService = new EmailService(); // Hard dependency
    this.database = new MySQLDatabase(); // Hard dependency
  }

  processOrder(order) {
    this.database.save(order);
    this.emailService.send(order.user, 'Order confirmed');
  }
}
```

Dependency Inversion - Good Example

```
// Abstractions
class DatabaseInterface {
  save(data) { throw new Error('Must implement'); }
}

class NotificationInterface {
  send(user, message) { throw new Error('Must implement'); }
}

// Concrete implementations
class MySQLDatabase extends DatabaseInterface {
  save(data) { /* MySQL implementation */ }
}

class EmailService extends NotificationInterface {
  send(user, message) { /* Email implementation */ }
}

// Service depends on abstractions
class OrderService {
  constructor(database, notificationService) {
    this.database = database; // Injected dependency
    this.notificationService = notificationService; // Injected dependency
  }

  processOrder(order) {
    this.database.save(order);
    this.notificationService.send(order.user, 'Order confirmed');
  }
}
```

Dependency Injection

Providing Dependencies from Outside

Types:

- **Constructor Injection** - Dependencies provided at creation
- **Setter Injection** - Dependencies set after creation
- **Interface Injection** - Dependencies provided through interface

Benefits:

- Easier testing with mocks
- Flexible configuration
- Reduced coupling

DI Container Example

```
// DI Container
class Container {
  constructor() {
    this.services = new Map();
  }

  register(name, factory) {
    this.services.set(name, factory);
  }

  resolve(name) {
    const factory = this.services.get(name);
    return factory ? factory() : null;
  }
}

// Registration
const container = new Container();
container.register('database', () => new MySQLDatabase());
container.register('emailService', () => new EmailService());
container.register('orderService', () =>
  new OrderService(
    container.resolve('database'),
    container.resolve('emailService')
  )
);

// Usage
const orderService = container.resolve('orderService');
```

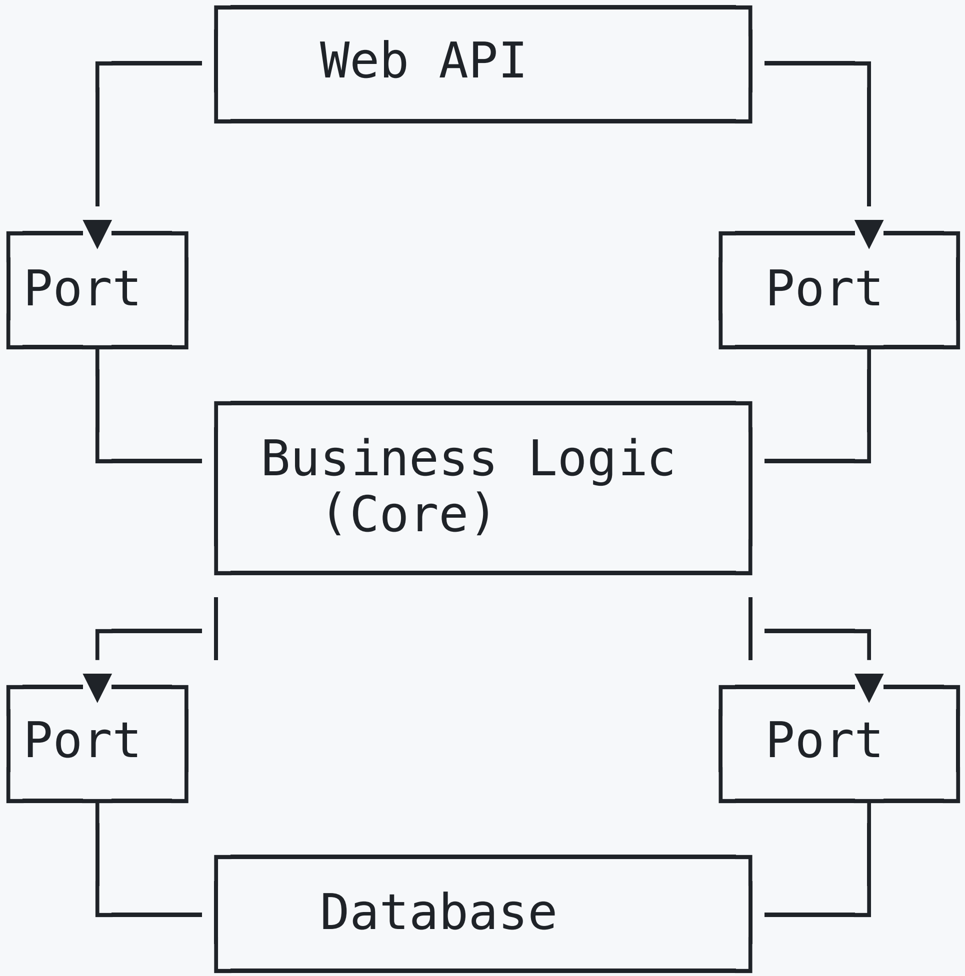
Separation of Concerns

Organizing Code by Responsibility

Layered Architecture:

```
┌──────────────────┐
│  Presentation    │  ← Controllers, Views
├──────────────────┤
│  Business Logic  │  ← Services, Domain Models
├──────────────────┤
│  Data Access     │  ← Repositories, DAOs
├──────────────────┤
│  Infrastructure  │  ← Database, External APIs
└──────────────────┘
```

Hexagonal Architecture (Ports & Adapters)



Module Design Patterns

Common Structural Patterns

Module Pattern:

```
const UserModule = (function() {
  // Private variables
  let users = [];

  // Private functions
  function validate(user) {
    return user.email && user.name;
  }

  // Public API
  return {
    add(user) {
      if (validate(user)) {
        users.push(user);
        return true;
      }
      return false;
    },
```

Factory Pattern

```
class DatabaseFactory {
  static create(type, config) {
    switch(type) {
      case 'mysql':
        return new MySQLDatabase(config);
      case 'postgresql':
        return new PostgreSQLDatabase(config);
      case 'mongodb':
        return new MongoDatabase(config);
      default:
        throw new Error(`Unknown database type: ${type}`);
    }
  }
}

// Usage
const db = DatabaseFactory.create('postgresql', {
  host: 'localhost',
  port: 5432,
  database: 'myapp'
});
```

Loose Coupling Strategies

Reducing Dependencies Between Modules

Event-Driven Architecture:

```
class EventBus {
  constructor() {
    this.listeners = new Map();
  }

  on(event, callback) {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event).push(callback);
  }

  emit(event, data) {
    const callbacks = this.listeners.get(event) || [];
```

```
// Usage
const eventBus = new EventBus();

// Modules listen for events
eventBus.on('user.created', (user) => {
  emailService.sendWelcomeEmail(user);
});

eventBus.on('user.created', (user) => {
  analyticsService.track('user_registered', user);
});

// Trigger events instead of direct calls
eventBus.emit('user.created', newUser);
```

High Cohesion

Related Functionality Together

Good Cohesion Example:

```
class UserValidator {
  validateEmail(email) {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
  }

  validatePassword(password) {
    return password.length >= 8 && /[A-Z]/.test(password);
  }

  validateAge(age) {
    return age >= 18 && age <= 120;
  }
```

```
  validateUser(user) {
    return this.validateEmail(user.email) &&
           this.validatePassword(user.password) &&
           this.validateAge(user.age);
  }
}
```

All methods related to user validation are together.

Testing Modular Code

Benefits of Modular Design for Testing

Unit Testing:

```
describe('UserValidator', () => {
  const validator = new UserValidator();

  test('validates email correctly', () => {
    expect(validator.validateEmail('test@example.com')).toBe(true);
    expect(validator.validateEmail('invalid-email')).toBe(false);
  });

  test('validates password correctly', () => {
    expect(validator.validatePassword('SecureP@ss')).toBe(true);
    expect(validator.validatePassword('weak')).toBe(false);
  });
});
```

Integration Testing with Mocks:

```
test('OrderService processes order correctly', () => {
  const mockDatabase = { save: jest.fn() };
  const mockEmailService = { send: jest.fn() };

  const orderService = new OrderService(mockDatabase, mockEmailService);
  const order = { id: 1, user: { email: 'test@example.com' } };

  orderService.processOrder(order);

  expect(mockDatabase.save).toHaveBeenCalledWith(order);
  expect(mockEmailService.send).toHaveBeenCalledWith(
    order.user,
    'Order confirmed'
  );
});
```

Documentation & Interfaces

Clear Module Contracts

Interface Documentation:

```
/**
 * User repository interface
 * @interface UserRepository
 */
class UserRepository {
  /**
   * Save a user to the data store
   * @param {User} user - The user to save
   * @returns {Promise<User>} The saved user with generated ID
   * @throws {ValidationError} When user data is invalid
   */
  async save(user) {
    throw new Error('Must be implemented by subclass');
  }
```

```
/**
 * Find user by email address
 * @param {string} email - User's email address
 * @returns {Promise<User|null>} User if found, null otherwise
 */
async findByEmail(email) {
  throw new Error('Must be implemented by subclass');
}
}
```

Package Organization

Structuring Modular Code

By Feature (Recommended):

```
src/
├── user/
│   ├── User.js
│   ├── UserRepository.js
│   ├── UserService.js
│   └── UserController.js
├── order/
│   ├── Order.js
│   ├── OrderRepository.js
│   ├── OrderService.js
│   └── OrderController.js
└── shared/
    ├── Database.js
    ├── EventBus.js
```

By Layer (Traditional):

```
src/
├── controllers/
├── services/
├── repositories/
└── models/
```

Module Boundaries

Defining Clear Interfaces

Public vs Private APIs:

```
// user/index.js - Public module interface
export { UserService } from './UserService.js';
export { User } from './User.js';

// Don't export internal implementation details
// UserRepository, UserValidator are private to this module
```

Cross-Module Communication:

```
// order/OrderService.js
import { UserService } from '../user/index.js';

class OrderService {
```

Refactoring to Modular Design

Incremental Improvement

Step 1: Identify Responsibilities

- What does each class/function do?
- Can responsibilities be separated?

Step 2: Extract Functions

```
// Before: Everything in one function
function processOrder(orderData) {
  // validation logic
  // database save logic
  // email sending logic
  // inventory update logic
}
```

Step 3: Create Abstractions

```
// Before: Direct dependencies
class OrderService {
  processOrder(orderData) {
    // Direct database calls
    const connection = mysql.createConnection(config);
    connection.query('INSERT INTO orders...', orderData);

    // Direct email sending
    nodemailer.sendMail({
      to: orderData.email,
      subject: 'Order Confirmation',
      text: 'Your order has been confirmed'
    });
  }
}

// After: Dependency injection
class OrderService {
  constructor(orderRepository, emailService) {
    this.orderRepository = orderRepository;
    this.emailService = emailService;
  }

  async processOrder(orderData) {
    const order = await this.orderRepository.save(orderData);
    await this.emailService.sendConfirmation(order);
    return order;
  }
}
```

Common Anti-Patterns

❌ What to Avoid:

God Objects:

- Classes that do too much
- Hundreds of lines of code
- Many responsibilities

Tight Coupling:

- Direct dependencies on concrete classes
- Hard-coded configuration
- Circular dependencies

Microservices Architecture

Ultimate Modular Design

Benefits:

- Independent deployment
- Technology diversity
- Team autonomy
- Fault isolation

Challenges:

- Network complexity
- Data consistency
- Service discovery

Key Takeaways

🎯 Remember:

1. **Single Responsibility** - Each module has one job
2. **Loose Coupling** - Minimize dependencies
3. **High Cohesion** - Related functionality together
4. **Clear Interfaces** - Well-defined contracts
5. **Dependency Injection** - Flexible, testable design

Implementation Checklist

✅ Building Modular Code:

- [ ] Apply SOLID principles to class design
- [ ] Use dependency injection for external dependencies
- [ ] Create clear interfaces and abstractions
- [ ] Organize code by feature, not layer
- [ ] Document module contracts and APIs
- [ ] Write unit tests for individual modules
- [ ] Minimize coupling between modules

Questions & Discussion

Discussion Points:

- What modular design challenges have you faced?
- How do you decide when to extract a new module?
- What patterns work best in your technology stack?
- How do you handle cross-cutting concerns?

Next Steps

Apply This Knowledge:

1. **Audit existing code** - Identify tightly coupled areas
2. **Extract functions** - Start with single responsibility
3. **Create interfaces** - Define clear contracts
4. **Implement DI** - Remove hard dependencies
5. **Write tests** - Verify module isolation
6. **Refactor incrementally** - Small, safe changes

Thank You

Next Topic:

Comprehensive Test Coverage

Ensuring reliability through testing strategies

Resources:

- SOLID Principles Guide
- Dependency Injection Patterns
- Refactoring Techniques