# miniCycle Modularization Guide v3.0

**The Multi-Pattern Approach: Choose the Right Tool for the Job**

---

## 🚨 Error Handling & Logging Standards

**Each pattern has specific error handling rules to keep console output clean and meaningful:**

| Pattern | Error Strategy | Logging Level | When to Throw |
|---|---|---|---|
| **Static Utility** ⚡ | Return safe defaults | `console.warn` only | Never |
| **Simple Instance** 🎯 | Graceful fallbacks | `console.warn` + fallback | Never |
| **Resilient Constructor** 🛡️ | Degrade gracefully | `console.warn` + user notification | Never |
| **Strict Injection** 🔧 | Fail fast | `throw` + `showNotification('error')` | Missing deps |

**Examples:**

javascript
```
// Static Utility - warn and return safe default
static safeGetElement(id) {
    const element = document.getElementById(id);
    if (!element) {
        console.warn(`⚠️ Element #${id} not found`);
        return null; // Safe default
    }
    return element;
}

// Simple Instance - warn and fallback
show(message, type) {
    try {
```

```
            this.createNotification(message, type);
        } catch (error) {
            console.warn('Notification error:', error);
            console.log(`[Fallback] ${message}`); // Always works
        }
    }

    // Resilient Constructor - warn and show user-visible fallback
    updateWidget() {
        try {
            const data = this.deps.loadData();
            this.renderWidget(data);
        } catch (error) {
            console.warn('Widget update failed:', error);
            this.deps.showNotification("Widget temporarily unavailable", "warning");
            this.showPlaceholderContent(); // User sees something
        }
    }

    // Strict Injection - throw with clear message
    function processData() {
        if (typeof Deps.loadData !== 'function') {
            throw new Error('dataProcessor: missing required dependency "loadData". Call
    setDataProcessorDependencies() first.');
        }
    }
```

---

# 🧪 Testing Strategy by Pattern

**Quick testing guidelines for each pattern:**

- **Static Utility** ⚡ → Pure unit tests, no mocks needed
- **Simple Instance** 🎯 → DOM smoke tests + fallback path verification
- **Resilient Constructor** 🛡️ → Dependency stubs + "missing dependency" scenarios
- **Strict Injection** 🔧 → Assertion tests (missing deps) + happy path + data persistence

javascript

```
// Example: Testing Simple Instance fallback
test('notification falls back to console when DOM unavailable', () => {
    // Simulate missing DOM
    document.getElementById = () => null;
```

```
    const consoleSpy = jest.spyOn(console, 'log');
    notifications.show('test message', 'info');

    expect(consoleSpy).toHaveBeenCalledWith('[Fallback] test message');
});
```

---

## ⚡ Proper Initialization Order

**Load modules in the right order to prevent race conditions:**

javascript
```
document.addEventListener('DOMContentLoaded', async () => {
    // 1) Static Utilities (no configuration needed)
    await import('./utilities/globalUtils.js');
    await import('./utilities/domHelpers.js');

    // 2) Simple Instances (ready immediately after import)
    await import('./utilities/notifications.js');

    // 3) Strict DI modules (configure BEFORE first use)
    const cycleLoader = await import('./utilities/cycleLoader.js');
    cycleLoader.setCycleLoaderDependencies({
        loadMiniCycleData: loadMiniCycleData,
        saveData: saveMiniCycleData,
        showNotification: showNotification,
        createElement: document.createElement.bind(document)
    });

    const dataProcessor = await import('./utilities/dataProcessor.js');
    dataProcessor.setDataProcessorDependencies({
        loadData: loadMiniCycleData,
        saveData: saveMiniCycleData,
        storage: window.localStorage,
        now: () => Date.now(),
        showNotification
    });

    // 4) Resilient UI Components (inject what's available, graceful if missing)
    const { StatsPanelManager } = await import('./utilities/statsPanel.js');
    const statsPanel = new StatsPanelManager({
        showNotification,
        loadData: loadMiniCycleData,
```

```
        isOverlayActive,
        updateThemeColor
    });

    console.log('✅ All modules initialized in proper order');
});
```

---

# ⚠️ Anti-Patterns: What NOT to Do

**Guardrails to prevent pattern misuse:**

## Static Utility ⚡ - Keep It Pure

javascript

```
// ❌ DON'T: Add state, storage, or DOM reads
static badUtility() {
    this.counter++; // ❌ No state
    localStorage.setItem('key', 'value'); // ❌ No storage
    const element = document.querySelector('.item'); // ❌ No DOM reads
}

// ✅ DO: Pure input → output transformations
static goodUtility(input) {
    return input.trim().toLowerCase(); // ✅ Pure function
}
```

## Simple Instance 🎯 - Don't Hide Critical Errors

javascript

```
// ❌ DON'T: Silently hide data corruption or critical failures
updateCriticalData() {
    try {
        this.performDataMigration();
    } catch (error) {
        // ❌ Silent failure could corrupt user data
        console.warn('Migration failed, ignoring...');
    }
}

// ✅ DO: Surface critical issues while providing fallbacks
updateCriticalData() {
    try {
```

```
        this.performDataMigration();
    } catch (error) {
        console.error('Data migration failed:', error);
        this.showNotification('Data update failed - please refresh', 'error');
        this.enableSafeMode(); // ✅ Visible degradation
    }
}
```

### Resilient Constructor 🛡️ - Don't Swallow Everything

```
javascript
// ❌ DON'T: Hide all errors from users
processUserAction() {
    try {
        this.performComplexOperation();
    } catch (error) {
        // ❌ User has no idea what happened
        console.warn('Something went wrong');
    }
}

// ✅ DO: Show users what's happening
processUserAction() {
    try {
        this.performComplexOperation();
        this.showNotification('Action completed', 'success');
    } catch (error) {
        console.warn('Operation failed:', error);
        this.showNotification('Action failed - using simplified mode', 'warning');
        this.enableSimplifiedMode(); // ✅ User understands the state
    }
}
```

### Strict Injection 🔧 - Don't Add Automatic Fallbacks

```
javascript
// ❌ DON'T: Provide fallbacks (defeats the purpose)
function processBusinessLogic() {
    const data = Deps.loadData || (() => ({})); // ❌ Hides config errors
    // ... business logic
}

// ✅ DO: Fail fast with helpful errors
function processBusinessLogic() {
```

```
    assertInjected('loadData', Deps.loadData); // ✅ Clear error if misconfigured
    const data = Deps.loadData();
    // ... business logic
}
```

---

# 🎯 Quick Start: Which Pattern Should I Use?

**Start here with your module type:**

```
Is it a foundational utility? (DOM helpers, formatters, validators)
├─ YES → Use Static Utility Pattern ⚡
└─ NO ↓

Does it have complex business logic with many dependencies?
├─ YES → Use Strict Dependency Injection Pattern 🔧
└─ NO ↓

Is it a complex UI component that must work even when things break?
├─ YES → Use Resilient Constructor Pattern 🛡️
└─ NO → Use Simple Instance Pattern 🎯
```

---

# 📚 The Four Proven Patterns

## ⚡ Pattern 1: Static Utility Pattern

✅ **Perfect for:** DOM helpers, formatters, validators, math functions, ID generators

✅ **When to use:** Pure utility functions with no external dependencies

✅ **Real example:** `globalUtils.js` in miniCycle

javascript

```
// utilities/domHelpers.js
export class DOMHelpers {
    /**
     * Safely add event listener, removing any existing one first
     */
    static safeAddEventListener(element, event, handler) {
        if (!element) return;
```

```
        element.removeEventListener(event, handler);
        element.addEventListener(event, handler);
    }

    /**
     * Get element with warning if not found
     */
    static safeGetElement(id, showWarning = true) {
        const element = document.getElementById(id);
        if (!element && showWarning) {
            console.warn(`⚠️ Element #${id} not found`);
        }
        return element;
    }

    /**
     * Generate unique IDs
     */
    static generateId(prefix = 'id') {
        return `${prefix}-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;
    }
}

// Make globally available for backward compatibility
window.safeAddEventListener = DOMHelpers.safeAddEventListener;
window.safeGetElement = DOMHelpers.safeGetElement;
window.generateId = DOMHelpers.generateId;

console.log('🛠️ DOM Helpers loaded - utilities available globally');
```

**Integration:**

javascript

```
// Just import and use immediately
import './utilities/domHelpers.js';

// Works right away
safeAddEventListener(button, 'click', handleClick);
const newId = generateId('task');
```

✅ **Advantages:** Zero setup, works everywhere, no configuration needed

---

## 🎯 Pattern 2: Simple Instance Pattern

✅ **Perfect for:** Notification systems, simple UI components, basic services

✅ **When to use:** Self-contained functionality that should gracefully degrade

✅ **Real example:** `notifications.js` in miniCycle

javascript

```javascript
// utilities/notifications.js
export class NotificationManager {
    constructor() {
        this.container = this.findOrCreateContainer();
        this.activeNotifications = new Set();
    }

    show(message, type = "info", duration = 3000) {
        try {
            const notification = this.createNotification(message, type);
            this.container.appendChild(notification);
            this.activeNotifications.add(notification);

            setTimeout(() => this.removeNotification(notification), duration);
            return notification;
        } catch (error) {
            // Graceful fallback
            console.log(`[Notification] ${message}`);
            console.warn('Notification system error:', error);
        }
    }

    findOrCreateContainer() {
        return document.getElementById('notification-container') ||
               document.body;
    }

    createNotification(message, type) {
        const notif = document.createElement('div');
        notif.className = `notification notification-${type}`;
        notif.textContent = message;
        return notif;
    }

    removeNotification(notification) {
        try {
```

```
            notification.remove();
            this.activeNotifications.delete(notification);
        } catch (error) {
            console.warn('Error removing notification:', error);
        }
    }
}

// Create instance with safe wrapper
const notifications = new NotificationManager();

function safeShowNotification(message, type = "info", duration = 3000) {
    try {
        return notifications.show(message, type, duration);
    } catch (error) {
        console.log(`[Fallback] ${message}`);
        console.warn('Notification error:', error);
    }
}

// Make globally available
window.showNotification = safeShowNotification;
window.notificationManager = notifications;

console.log('🔔 Notification system loaded and ready');
```

**Integration:**

```
javascript
// Import and it works immediately
import './utilities/notifications.js';

// Use right away
showNotification("Task completed!", "success");
showNotification("Warning message", "warning", 5000);
```

✅ **Advantages:** Works out of the box, graceful error handling, self-contained

---

## 🛡️ Pattern 3: Resilient Constructor Pattern

✅ **Perfect for:** Complex UI components, interactive panels, dashboard widgets

✅ **When to use:** Components needing external functions but must work when they're missing

✅ **Real example:** `statsPanel.js` in miniCycle

javascript

```javascript
// utilities/complexWidget.js
export class ComplexWidget {
    constructor(dependencies = {}) {
        // Store dependencies with intelligent fallbacks
        this.deps = {
            showNotification: dependencies.showNotification || this.fallbackNotification,
            loadData: dependencies.loadData || this.fallbackLoadData,
            saveSettings: dependencies.saveSettings || this.fallbackSaveSettings,
            isOverlayActive: dependencies.isOverlayActive || (() => false)
        };

        // Internal state
        this.state = {
            isVisible: false,
            data: null,
            lastUpdate: null
        };

        // Initialize
        this.init();
    }

    /**
     * Main functionality - handles errors gracefully
     */
    updateWidget() {
        try {
            const data = this.deps.loadData();
            if (data) {
                this.state.data = data;
                this.state.lastUpdate = new Date();
                this.renderWidget(data);
                this.deps.showNotification("Widget updated", "success", 2000);
            } else {
                this.showNoDataState();
            }
        } catch (error) {
            console.warn('Widget update failed:', error);
            this.showErrorState();
```

```
    }
}

saveUserSettings(settings) {
    try {
        this.deps.saveSettings('widgetSettings', settings);
        this.deps.showNotification("Settings saved", "success");
    } catch (error) {
        console.warn('Failed to save settings:', error);
        this.deps.showNotification("Could not save settings", "warning");
    }
}

// Fallback methods
fallbackNotification(message, type) {
    console.log(`[Widget] ${message}`);
}

fallbackLoadData() {
    console.warn('⚠️ Data loading not available - showing placeholder');
    return { placeholder: true, message: 'Data unavailable' };
}

fallbackSaveSettings(key, value) {
    console.warn('⚠️ Settings save not available - using localStorage fallback');
    try {
        localStorage.setItem(key, JSON.stringify(value));
    } catch (e) {
        console.warn('Even localStorage failed:', e);
    }
}

init() {
    console.log('🎛️ Complex widget initializing...');
    this.updateWidget();
}

renderWidget(data) {
    // Render logic here
    console.log('Widget rendered with data:', data);
}

showNoDataState() {
    this.deps.showNotification("No data available", "info");
```

```
    }

    showErrorState() {
        this.deps.showNotification("Widget error - using fallback mode", "warning");
    }
}

// Global management
let complexWidget = null;

function updateComplexWidget() {
    return complexWidget?.updateWidget();
}

function saveWidgetSettings(settings) {
    return complexWidget?.saveUserSettings(settings);
}

// Make globally available
window.updateComplexWidget = updateComplexWidget;
window.saveWidgetSettings = saveWidgetSettings;
```

**Integration:**

javascript

```
// Import and configure
const { ComplexWidget } = await import('./utilities/complexWidget.js');

// Initialize with available dependencies
complexWidget = new ComplexWidget({
    showNotification: window.showNotification,
    loadData: window.loadMiniCycleData,
    saveSettings: window.saveUserSetting,
    isOverlayActive: window.isOverlayActive
});

// Widget works even if some dependencies are missing
```

✅ **Advantages:** Resilient to missing dependencies, graceful degradation, helpful error messages

---

## 🔧 Pattern 4: Strict Dependency Injection Pattern

✅ **Perfect for:** Complex business logic, data processing, critical app functionality

✅ **When to use:** Mission-critical functionality that CANNOT work without dependencies

✅ **Real example:** `cycleLoader.js` in miniCycle

javascript

```javascript
// utilities/dataProcessor.js

// Define required dependencies
const Deps = {
    loadData: null,
    saveData: null,
    showNotification: null,
    validateData: null,
    createElement: null,
    formatDate: null
};

/**
 * Set up dependencies before using module
 */
function setDataProcessorDependencies(overrides = {}) {
    Object.assign(Deps, overrides);
    console.log('📦 DataProcessor dependencies configured');
}

/**
 * Ensure dependency is available
 */
function assertInjected(name, fn) {
    if (typeof fn !== 'function') {
        throw new Error(`dataProcessor: missing required dependency '${name}'. Call
setDataProcessorDependencies() first.`);
    }
}

/**
 * Process a batch of tasks - requires all dependencies
 */
export function processTaskBatch(tasks, options = {}) {
    assertInjected('loadData', Deps.loadData);
    assertInjected('saveData', Deps.saveData);
```

```
assertInjected('showNotification', Deps.showNotification);
assertInjected('validateData', Deps.validateData);

try {
    // Load current data
    const currentData = Deps.loadData();
    if (!currentData) {
        throw new Error('No data available to process');
    }

    // Validate input
    const validTasks = tasks.filter(task => {
        const isValid = Deps.validateData(task);
        if (!isValid) {
            console.warn('Invalid task filtered out:', task);
        }
        return isValid;
    });

    if (validTasks.length === 0) {
        throw new Error('No valid tasks to process');
    }

    // Process tasks
    const processed = validTasks.map(task => ({
        ...task,
        processed: true,
        processedAt: new Date().toISOString(),
        processingOptions: options
    }));

    // Save results
    const updatedData = { ...currentData, tasks: processed };
    Deps.saveData(updatedData);

    // Notify success
    Deps.showNotification(
        `✅ Successfully processed ${processed.length} tasks`,
        'success',
        3000
    );

    return processed;
```

```
    } catch (error) {
        Deps.showNotification(
            `❌ Processing failed: ${error.message}`,
            'error',
            5000
        );
        throw error; // Re-throw for caller to handle
    }
}

/**
 * Advanced data analysis - also requires dependencies
 */
export function analyzeTaskData(analysisType = 'basic') {
    assertInjected('loadData', Deps.loadData);
    assertInjected('formatDate', Deps.formatDate);

    const data = Deps.loadData();
    const tasks = data?.tasks || [];

    const analysis = {
        totalTasks: tasks.length,
        completedTasks: tasks.filter(t => t.completed).length,
        analysisDate: Deps.formatDate(new Date()),
        analysisType
    };

    if (analysisType === 'detailed') {
        analysis.tasksByCategory = tasks.reduce((acc, task) => {
            const category = task.category || 'uncategorized';
            acc[category] = (acc[category] || 0) + 1;
            return acc;
        }, {});
    }

    return analysis;
}

// Export the setup function
export { setDataProcessorDependencies };
```

**Integration:**

javascript

```
// Import module
const dataMod = await import('./utilities/dataProcessor.js');

// MUST configure dependencies before use
dataMod.setDataProcessorDependencies({
    loadData: loadMiniCycleData,
    saveData: saveMiniCycleData,
    showNotification: showNotification,
    validateData: validateTaskData,
    createElement: document.createElement.bind(document),
    formatDate: formatDateString
});

// Now safe to use - will fail with clear errors if misconfigured
try {
    const processed = dataMod.processTaskBatch(selectedTasks, { priority: 'high' });
    const analysis = dataMod.analyzeTaskData('detailed');
    console.log('Processing completed:', processed, analysis);
} catch (error) {
    console.error('Processing failed:', error.message);
}
```

✅ **Advantages:** Crystal clear dependencies, fail-fast with helpful errors, highly testable

---

## 🏠 Real miniCycle Examples

**See these patterns in action in your codebase:**

- `utilities/globalUtils.js` → **Static Utility** ⚡ (DOM helpers, formatters)
- `utilities/notifications.js` → **Simple Instance** 🎯 (notification system + educational tips)
- `utilities/statsPanel.js` → **Resilient Constructor** 🛡️ (stats panel with swipe detection)
- `utilities/cycleLoader.js` → **Strict Injection** 🔧 (cycle loading with explicit dependencies)

---

## 🔄 Global Wrapper Policy

**Purpose:** Maintain backward compatibility during migration while providing a clear upgrade path.

**The Rule:** Global wrappers are temporary bridges that:

- Provide thin pass-through calls to module APIs
- Log deprecation warnings (once per session)
- Get removed after several versions

javascript

```
// utilities/globals.js (loaded after all modules)
import { NotificationManager } from './utilities/notifications.js';
const _notifications = new NotificationManager();
let _deprecationWarnings = new Set();

function _warnOnce(functionName) {
    if (!_deprecationWarnings.has(functionName)) {
        console.warn(`[Deprecation] Global ${functionName}() will be removed. Use ES6 imports instead.`);
        _deprecationWarnings.add(functionName);
    }
}

// Temporary global wrapper (remove in v2.0)
window.showNotification = (msg, type, dur) => {
    _warnOnce('showNotification');
    return _notifications.show(msg, type, dur);
};

// Modern usage (encourage this):
// import { NotificationManager } from './utilities/notifications.js';
// const notifications = new NotificationManager();
```

---

# 📋 Naming Conventions for Dependency Injection

**Setup Functions:** Always `set<ModuleName>Dependencies(overrides)`

javascript

```
setNotificationDependencies()   ✅
setStatsPanelDependencies()     ✅
setupNotificationDeps()         ❌ (inconsistent)
configureNotifications()        ❌ (unclear purpose)
```

**Common Dependency Names:** Use these standard names to prevent drift

```javascript
const Deps = {
  // Data operations
  loadData: null,       // Load app data
  saveData: null,       // Save app data
  storage: null,        // Direct storage access

  // UI feedback
  showNotification: null,// Show user notifications
  logger: null,         // Console/debug logging

  // Utilities
  now: null,            // () => Date.now() for testing
  createElement: null,  // document.createElement.bind(document)

  // App-specific
  getCurrentUser: null,  // Get current user context
  isOverlayActive: null  // Check if modal/overlay open
};
```

**Don't inject `document` or `window` - inject specific functions instead:**

```javascript
// ❌ Too broad
createElement: document

// ✅ Specific and testable
createElement: document.createElement.bind(document)
```

---

# 🔄 Module Lifecycle Standards

**Consistent lifecycle methods for UI patterns:**

```javascript
// Simple Instance Pattern
class SimpleUIComponent {
  constructor() { /* setup */ }
  destroy() { /* cleanup - optional */ }
}
```

```
// Resilient Constructor Pattern
class ComplexUIComponent {
    constructor(deps) { /* setup with fallbacks */ }
    init() { /* initialize after construction - no throws */ }
    update() { /* refresh data/state - no throws */ }
    destroy() { /* cleanup resources - no throws */ }
}
```

**Every UI module exports these standard methods when applicable:**

- `init()` - Initialize after dependencies are ready
- `update()` - Refresh or re-render content
- `destroy()` - Clean up event listeners and resources

---

## 📊 Pattern Selection Reference

| Module Type | Best Pattern | Key Indicators |
|---|---|---|
| **DOM Utilities** | Static Utility ⚡ | Pure functions, no state, universal |
| **Math Functions** | Static Utility ⚡ | Input → output, no side effects |
| **Formatters** | Static Utility ⚡ | Transform data, no dependencies |
| **Notifications** | Simple Instance 🎯 | Self-contained, should always work |
| **Simple Modals** | Simple Instance 🎯 | Basic UI, graceful degradation |
| **Status Panels** | Resilient Constructor 🛡️ | Complex UI, needs external data |
| **Interactive Widgets** | Resilient Constructor 🛡️ | Must handle missing dependencies |
| **Data Processing** | Strict Injection 🔧 | Critical logic, complex dependencies |

| | | |
|---|---|---|
| **Core App Features** | Strict Injection 🔧 | Cannot work without dependencies |

---

# 🛠️ Implementation Checklist

## Before You Start:

- Identify what your module does (use decision tree)
- List all external functions/data it needs
- Decide if missing dependencies should be fatal or handled gracefully
- Choose the appropriate pattern

## For Every Pattern:

- Create clean, minimal exports
- Add console.log for successful loading
- Create global wrapper functions for backward compatibility
- Test with missing dependencies to verify error handling

## Pattern-Specific Tasks:

**Static Utility ⚡:**

- All methods are static
- No constructor needed
- No external dependencies
- Functions are pure (same input = same output)

**Simple Instance 🎯:**

- Single constructor call creates working instance
- Built-in try/catch with fallbacks
- Console warnings for problems, not errors
- Works even when DOM elements are missing

**Resilient Constructor 🛡️:**

- Dependencies parameter with fallback functions
- Each major function has error handling
- Fallback methods provide reasonable alternatives
- State management for internal data

**Strict Injection 🔧:**

- Dependencies object clearly defined
- Setup function for dependency injection
- Assertion helper with clear error messages
- All external access goes through injected functions

---

# 🎓 Lessons Learned from Real Implementation

## ✅ What Works Well:

### 1. Match Pattern to Purpose

- Static utilities for simple, pure functions
- Simple instances for "fire and forget" functionality
- Resilient constructors for complex UI that must be robust
- Strict injection for critical business logic

### 2. Error Handling Strategy

- Static utilities: Return safe defaults or null
- Simple instances: Console warnings + fallback behavior
- Resilient constructors: Graceful degradation with user feedback
- Strict injection: Fail fast with clear error messages

### 3. Global Compatibility

- Always provide global wrapper functions
- Maintains backward compatibility during migration
- Allows gradual adoption of modular patterns

## ⚠️ Common Pitfalls:

### 1. Wrong Pattern Choice

- Don't use strict injection for simple utilities
- Don't use static methods for stateful components
- Don't use simple instances for critical business logic

### 2. Over-Engineering

- Keep static utilities truly static
- Don't add dependencies to things that don't need them
- Simple is better when it works

**3. Under-Engineering**

- Don't skip error handling
- Don't assume dependencies will always be available
- Don't forget to test failure modes

---

# 🚀 Migration Strategy

## Phase 1: Start With Static Utilities

Easiest wins with immediate benefits:

- DOM helper functions
- Formatters and validators
- Math and string utilities

## Phase 2: Extract Simple Services

Self-contained functionality:

- Notification systems
- Basic modal management
- Simple data storage helpers

## Phase 3: Modularize Complex UI

Interactive components:

- Statistics panels
- Settings interfaces
- Complex form handlers

## Phase 4: Core Business Logic

Mission-critical functionality:

- Data processing engines
- Complex calculations
- Multi-step workflows

---

# 🔍 Troubleshooting Guide

### "Module not working after import"

- **Static Utility:** Check if functions are available globally
- **Simple Instance:** Look for error messages in console
- **Resilient Constructor:** Verify it initialized without errors
- **Strict Injection:** Ensure you called the setup function

### "Getting dependency errors"

- **Simple Instance:** This is normal, check fallback behavior
- **Resilient Constructor:** Expected, verify graceful degradation
- **Strict Injection:** This is intentional - configure dependencies first

### "Functions not globally available"

- Check that `window.functionName =` assignments are present
- Verify the module import actually executed
- Look for console messages confirming module loaded

### "Module works but app doesn't"

- Check if old code is calling functions that moved
- Verify global wrapper functions are working
- Test individual module functions in browser console

---

# ✅ Success Indicators

You'll know you chose the right pattern when:

- **Static Utilities:** Work everywhere immediately, no configuration needed
- **Simple Instances:** Keep working even when other systems break
- **Resilient Constructors:** Degrade gracefully, show helpful warnings
- **Strict Injection:** Fail fast with crystal-clear error messages when misconfigured

---

# 🎯 Updated Quick Start Template

javascript

```
// 1. Choose your pattern using the decision tree above
// 2. Copy the appropriate pattern example
// 3. Follow the naming conventions for DI setup functions
// 4. Add proper error handling for your pattern type
// 5. Include lifecycle methods (init/update/destroy) for UI components
// 6. Test error conditions and fallback behaviors
// 7. Add global wrappers with deprecation warnings
// 8. Import in proper initialization order

// Example: Complete module integration in main script
document.addEventListener('DOMContentLoaded', async () => {
    console.log('🚀 Initializing miniCycle modules...');

    // Phase 1: Static utilities (instant, no config)
    await import('./utilities/globalUtils.js');
    console.log('⚡ Static utilities loaded');

    // Phase 2: Simple services (instant, self-configuring)
    await import('./utilities/notifications.js');
    console.log('🎯 Simple services ready');

    // Phase 3: Strict DI modules (must configure first!)
    const processor = await import('./utilities/dataProcessor.js');
    processor.setDataProcessorDependencies({
        loadData: loadMiniCycleData,
        saveData: saveMiniCycleData,
        storage: window.localStorage,
        now: () => Date.now(),
        showNotification
    });
    console.log('🔧 Business logic configured');

    // Phase 4: Resilient UI (graceful degradation built-in)
    const { ComplexWidget } = await import('./utilities/complexWidget.js');
    const widget = new ComplexWidget({
        showNotification,
        loadData: loadMiniCycleData,
        isOverlayActive
    });
    await widget.init();
    console.log('🛡️ UI components initialized');

    console.log('✅ All modules loaded successfully');
});
```

---

## 🎓 Key Takeaways

**Your miniCycle codebase proves these patterns work in production:**

1. **Different modules need different approaches** - not everything should use the same pattern
2. **Static utilities should stay pure** - no state, no dependencies, just input → output
3. **Simple instances should gracefully degrade** - always provide a fallback that works
4. **Complex UI should be resilient** - handle missing dependencies with user-visible degradation
5. **Critical business logic should fail fast** - missing dependencies should throw clear errors
6. **Global wrappers ease migration** - but mark them for eventual removal
7. **Consistent naming prevents confusion** - use standard dependency names across modules
8. **Initialize in order** - utilities first, then services, then configured modules, then UI

---

**Remember:** These patterns aren't theoretical - they're proven in your production miniCycle app. You've already successfully implemented each pattern for different purposes:

- **globalUtils.js** shows how Static Utilities provide zero-config foundation functions
- **notifications.js** demonstrates how Simple Instances work immediately with graceful fallbacks
- **statsPanel.js** proves Resilient Constructors can handle missing dependencies elegantly
- **cycleLoader.js** validates that Strict Injection ensures critical code gets what it needs

This guide simply helps you **apply the right pattern to each new module** as you continue modernizing your codebase. **Choose the pattern that fits the job, not the other way around.**