# Testing Strategy for Novellium

## Goal

Achieve 95% code coverage across all critical components of the Novellium platform.

## Testing Framework

We recommend using **Jest** for JavaScript testing due to its:

- Built-in code coverage reporting
- Easy setup with ES6 modules
- Snapshot testing capabilities
- Mocking and spying features

## Installation

```
npm init -y
npm install --save-dev jest @jest/globals jsdom
```

## Configuration

Create `jest.config.js`:

```
module.exports = {
  testEnvironment: 'jsdom',
  coverageThreshold: {
    global: {
      branches: 95,
      functions: 95,
      lines: 95,
      statements: 95
    }
  },
  collectCoverageFrom: [
    'modules/**/*.js',
    '!modules/**/*.test.js',
    '!node_modules/**'
  ],
  testMatch: [
    '**/tests/**/*.test.js'
  ],
  moduleNameMapper: {
    '^@/(.*)$': '<rootDir>/$1'
  }
};
```

Add to `package.json`:

```
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage",
    "test:ci": "jest --ci --coverage --maxWorkers=2"
  }
}
```

## Test Structure

```
visual-novel-engine/
├── tests/
│   ├── setup.md (this file)
│   ├── unit/
│   │   ├── SaveManager.test.js
│   │   ├── AssetLoader.test.js
│   │   ├── ConditionEvaluator.test.js
│   │   ├── Character.test.js
│   │   └── Event.test.js
│   ├── integration/
│   │   ├── gameLoading.test.js
│   │   ├── saveLoad.test.js
│   │   ├── deployment.test.js
│   │   └── builder.test.js
│   ├── e2e/
│   │   ├── fullGamePlaythrough.test.js
│   │   ├── buildAndDeploy.test.js
│   │   └── settingsCustomization.test.js
│   └── fixtures/
│       ├── testGame.json
│       ├── testSave.vnsave
│       └── mockAssets.js
│
```

## Testing Priority by Component

### 1. Core Engine (HIGH PRIORITY - Target 98% Coverage)

**VisualNovelEngine.js**

- Event execution flow
- Choice handling
- Variable management
- Background/character rendering

- Audio playback
- Typewriter effect
- Error handling

**Tests needed:**

- Test dialogue events display correctly
- Test narration events display correctly
- Test choice events present options
- Test event chaining (next property)
- Test variable setting and retrieval
- Test conditional event execution
- Test background changes
- Test character sprite changes
- Test audio playback/stop
- Test typewriter effect timing
- Test error handling for missing events

## 2. Save System (HIGH PRIORITY - Target 98% Coverage)

**SaveManager.js**

- Save creation with metadata
- Save loading
- Auto-save functionality
- Export/Import (.vnsave files)
- Auto-backup system
- Save deletion
- Save listing

**Tests needed:**

- Test creating saves with valid data
- Test loading saves restores state
- Test auto-save creates saves automatically
- Test export creates valid .vnsave file
- Test import loads .vnsave file correctly
- Test backup creation on save
- Test invalid save handling
- Test save cleanup (old saves)
- Test save slots (manual vs auto)

## 3. Asset Management (MEDIUM PRIORITY - Target 95% Coverage)

**AssetLoader.js**

- IndexedDB initialization
- Asset upload (images, audio)
- Asset retrieval

- Base64 encoding/decoding
- Asset deletion
- Asset preview generation

**Tests needed:**

- Test IndexedDB connection
- Test asset upload success
- Test asset retrieval by ID
- Test asset listing
- Test asset deletion
- Test file type validation
- Test large file handling
- Test concurrent uploads

### 4. Builder System (MEDIUM PRIORITY - Target 95% Coverage)

**build.html JavaScript**

- Project data management
- Character creation/editing
- Event creation/editing
- Form validation
- JSON editor mode
- Export to ZIP
- Import & Deploy validation

**Tests needed:**

- Test project info saving
- Test character CRUD operations
- Test event CRUD operations
- Test form-to-JSON conversion
- Test JSON-to-form conversion
- Test ZIP export structure
- Test game validation before deploy
- Test deployment to localStorage

### 5. Game Loading (HIGH PRIORITY - Target 98% Coverage)

**loadGamesList() and related functions**

- JSON fetching
- Deployed games merging
- Error handling for missing games
- Game selection
- Thumbnail loading fallback

**Tests needed:**

- Test fetching games-list.json
- Test merging deployed games
- Test handling missing JSON
- Test game selection updates UI
- Test thumbnail fallback
- Test empty game list handling

### 6. UI State Management (MEDIUM PRIORITY - Target 90% Coverage)

**Start Screen, Navigation, Settings**

- showStartScreen() rendering
- Navigation between screens
- Settings persistence
- CSS customization
- Dynamic style injection

**Tests needed:**

- Test start screen renders correctly
- Test navigation button clicks
- Test settings save to localStorage
- Test CSS color customization
- Test dynamic style injection
- Test responsive layout

## Test Examples

### Example 1: SaveManager Unit Test

```
// tests/unit/SaveManager.test.js
import SaveManager from '../../modules/SaveManager.js';

describe('SaveManager', () => {
  let saveManager;

  beforeEach(() => {
    saveManager = new SaveManager('test-game');
    localStorage.clear();
  });

  test('should create a new save', () => {
    const saveData = {
      currentEventId: 'chapter1',
      variables: { score: 10 },
      characterStates: {}
    };

    const save = saveManager.createSave(saveData, 1, 'manual');
    expect(save).toHaveProperty('id');
    expect(save.gameData).toEqual(saveData);
    expect(save.type).toBe('manual');
    expect(save.slot).toBe(1);
  });

  test('should load an existing save', () => {
    const saveData = {
      currentEventId: 'chapter1',
      variables: { score: 10 },
      characterStates: {}
    };

    const save = saveManager.createSave(saveData, 1, 'manual');
    const loaded = saveManager.loadSave(save.id);

    expect(loaded).toEqual(save);
  });

  test('should export save as .vnsave file', () => {
    const saveData = {
      currentEventId: 'chapter1',
      variables: { score: 10 },
      characterStates: {}
    };

    const save = saveManager.createSave(saveData, 1, 'manual');
    const exported = saveManager.exportSave(save.id);

    expect(exported).toContain('"gameId":"test-game"');
    expect(exported).toContain('"currentEventId":"chapter1"');
  });
});
```

Example 2: Game Loading Integration Test

```
// tests/integration/gameLoading.test.js
import { loadGame } from '../../index.js';

describe('Game Loading Integration', () => {
  beforeEach(() => {
    document.body.innerHTML = `
      <div id="game-container"></div>
      <div id="dialogue-box"></div>
      <div id="character-container"></div>
    `;
  });

  test('should load a complete game', async () => {
    const gameFolder = 'test-game';

    await loadGame(gameFolder);

    expect(engine).toBeDefined();
    expect(currentGame).toBe(gameFolder);
    expect(document.getElementById('game-
container').style.display).toBe('block');
  });

  test('should handle missing game gracefully', async () => {
    const gameFolder = 'nonexistent-game';

    await expect(loadGame(gameFolder)).rejects.toThrow();
  });
});
```

Example 3: Builder Validation Test

```
// tests/integration/builder.test.js
import { validateGameData } from '../../build.html';

describe('Builder Validation', () => {
  test('should validate complete game data', () => {
    const gameData = {
      info: {
        title: 'Test Game',
        id: 'test-game',
        author: 'Test Author',
        initialEvent: 'start'
      },
      characters: [
        {
          id: 'narrator',
          name: 'Narrator',
          color: '#ffffff',
          sprites: { default: 'narrator.png' }
        }
      ],
      events: [
        {
          type: 'dialogue',
          id: 'start',
          character: 'narrator',
          text: 'Welcome!',
          next: 'end'
        }
      ]
    };
    const result = validateGameData(gameData);

    expect(result.valid).toBe(true);
    expect(result.errors).toHaveLength(0);
  });

  test('should catch missing initial event', () => {
    const gameData = {
      info: { title: 'Test', id: 'test', initialEvent: 'missing' },
      characters: [],
      events: []
    };

    const result = validateGameData(gameData);

    expect(result.valid).toBe(false);
    expect(result.errors).toContain('Initial event "missing" not found');
  });
});
```

## Continuous Integration

### GitHub Actions Workflow

Create `.github/workflows/test.yml`:

```
name: Test Coverage

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'

    - name: Install dependencies
      run: npm ci

    - name: Run tests with coverage
      run: npm run test:ci

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3
```

```
        with:
          file: ./coverage/coverage-final.json

      - name: Check coverage threshold
        run: |
          COVERAGE=$(cat coverage/coverage-summary.json | jq '.total.lines.pct')
          if (( $(echo "$COVERAGE < 95" | bc -l) )); then
            echo "Coverage $COVERAGE% is below 95% threshold"
            exit 1
          fi
```

## Running Tests

### During Development

```
npm run test:watch
```

### Before Commit

```
npm run test:coverage
```

### In CI/CD

```
npm run test:ci
```

## Coverage Reports

After running tests with coverage:

1. Open `coverage/lcov-report/index.html` in browser
2. Review uncovered lines
3. Add tests for missed edge cases
4. Re-run until 95%+ coverage achieved

## Testing Checklist

- [ ] Unit tests for all modules (SaveManager, AssetLoader, etc.)
- [ ] Integration tests for game loading flow
- [ ] Integration tests for save/load cycle
- [ ] Integration tests for builder deployment
- [ ] E2E test for full game playthrough
- [ ] E2E test for build-and-deploy workflow

- [ ] Test fixtures created (mock games, saves, assets)
- [ ] Coverage threshold configured (95%)
- [ ] CI/CD pipeline setup
- [ ] Coverage badge added to README

## Next Steps

1. Install Jest and dependencies
2. Create test files based on priority order
3. Start with high-priority components (Engine, SaveManager)
4. Run coverage reports and identify gaps
5. Write tests for uncovered code
6. Setup CI/CD pipeline
7. Maintain tests as features are added

## Maintenance

- Run tests before every commit
- Update tests when features change
- Review coverage reports weekly
- Add regression tests for bugs
- Keep test fixtures up to date