SKILL

1 CC2.0 CREATE Function Skill

Function: CREATE - Code Generation & Implementation **Category Theory**: Free Monad + Coalgebra (Pure, Free, unfold) **Purpose**: Transform strategic plans into concrete implementations with quality tracking **Status**: 🔶 90% Complete (refactoring in progress, functional)

1.1 When to Use This Skill

Use CREATE when you need to: - 💻 **Code generation** - Transform plans into TypeScript/JavaScript code - 📝 **Documentation generation** - Create README, API docs, comments - 🧪 **Test generation** - Generate unit, integration, e2e tests - 🎨 **Template-based creation** - Use learned patterns for consistent output - 🔄 **Quality feedback loops** - Iterative refinement until threshold met

---

1.2 Core Capabilities

1.2.1 1. Code Generation (Free Monad)

```
// Build complex generation from simple primitives
const code = Pure(plan)
  .bind(generateStructure)
  .bind(addTypes)
  .bind(addDocumentation)
  .bind(addTests);
```

1.2.2 2. Pattern → Code (Coalgebra Unfold)

```
// Unfold pattern into code
const implementation = unfold(pattern, {
  template: "class-with-methods",
  context: { className: "UserService", methods: [...] }
});
```

1.2.3 3. Quality-Driven Refinement

```
// Iterative improvement until quality threshold
let artifact = generateInitial(plan);
let quality = await measureQuality(artifact);

while (quality < threshold && iterations < maxIterations) {
  artifact = refine(artifact, quality.issues);
  quality = await measureQuality(artifact);
  iterations++;
}
```

1.3 Input/Output Format

1.3.1 Input (from REASON)

```json
{
  "plan": {
    "strategy": "design",
    "steps": ["create class", "add methods", "add tests"],
    "context": {
      "className": "UserService",
      "methods": ["create", "read", "update", "delete"]
    }
  },
  "constraints": {
    "language": "typescript",
    "qualityThreshold": 0.80,
    "maxIterations": 3
  }
}
```

1.3.2 Output

```json
{
  "artifact": {
    "code": "...",
    "tests": "...",
    "documentation": "..."
  },
  "quality": {
    "score": 0.87,
    "dimensions": {
      "correctness": 0.90,
      "readability": 0.85,
      "maintainability": 0.88,
      "performance": 0.85,
      "security": 0.90
    },
    "issues": [],
    "antiPatterns": []
  },
  "iterations": 2,
  "confidence": 0.89
}
```

1.4 Generation Strategies

1.4.1 1. TypeScript Strategy

Generates: Classes, functions, interfaces, types **Quality Focus**: Type safety, null handling, error boundaries **Templates**: Available for common patterns (services, controllers, utilities)

1.4.2 2. Documentation Strategy

Generates: README.md, JSDoc comments, API references **Quality Focus**: Clarity, completeness, examples **Templates**: Project README, function docs, module docs

1.4.3 3. Testing Strategy

Generates: Jest/Vitest tests (unit, integration, e2e) **Quality Focus**: Coverage, edge cases, mocking **Templates**: Unit test suites, integration scenarios

---

1.5 Integration with Meta-Infrastructure

1.5.1 Quality Analyzer Agent

```
// Measure generated code quality
const quality = await qualityAnalyzer.analyze(code, {
  language: 'typescript',
  purpose: 'User service implementation'
});

if (quality.overallScore < 0.80) {
  // Trigger refinement
  code = await refine(code, quality.issues);
}
```

1.5.2 Template Optimizer

```
// Learn from successful generations
await templateOptimizer.recordUsage(template.id, {
  quality: quality.score,
  context: generation.context,
  successful: quality.score >= 0.80
});

// Optimize template over time
if (usageCount % 20 === 0) {
  const optimization = await templateOptimizer.optimize(
    template,
    usageHistory
  );
}
```

1.5.3 Anti-Pattern Detector

```
// Prevent bad code
const antiPatterns = await antiPatternDetector.detect(
  code,
  'typescript'
);

if (antiPatterns.some(p => p.severity === 'critical')) {
  // Regenerate or refine
  code = await regenerateAvoiding(antiPatterns);
}
```

1.6 Practical Examples

1.6.1 Example 1: Service Class Generation

```
// Input (from REASON)
const plan = {
  strategy: "design",
  steps: ["create service class", "add CRUD methods", "add validation"],
  context: {
    className: "UserService",
    methods: ["create", "read", "update", "delete"],
    validation: true
  }
};

// Execute CREATE
const result = await create(plan);

// Output
{
  code: `
export class UserService {
  async create(user: User): Promise<User> {
    // Validation
    if (!user.email || !user.name) {
      throw new ValidationError('Email and name required');
    }

    // Create user
    return await this.db.users.create(user);
  }

  async read(id: string): Promise<User | null> {
    return await this.db.users.findById(id);
  }

  // ... update, delete
}
  `,
  quality: {
    score: 0.87,
    dimensions: {
      correctness: 0.90,
      readability: 0.85,
      maintainability: 0.88
    }
  },
  iterations: 1
}
```

1.6.2 Example 2: Test Generation

```
// Input
const plan = {
  strategy: "testing",
  context: {
    targetCode: userServiceCode,
    testType: "unit",
    coverage: ["happy path", "error cases"]
  }
};

// Execute CREATE
const result = await create(plan);

// Output
{
  code: `
describe('UserService', () => {
  let service: UserService;

  beforeEach(() => {
    service = new UserService(mockDb);
  });

  describe('create', () => {
    it('should create user with valid data', async () => {
      const user = { email: 'test@example.com', name: 'Test' };
      const result = await service.create(user);
      expect(result).toMatchObject(user);
    });

    it('should throw ValidationError with missing email', async () => {
      const user = { name: 'Test' };
      await expect(service.create(user)).rejects.toThrow(ValidationError);
    });
  });
});
  `,
  quality: { score: 0.85 }
}
```

1.7 Quality Dimensions

CREATE tracks 5 quality dimensions:

1. **Correctness** (0-1): Does the code work as intended?
    - Type safety, logic correctness, edge case handling
2. **Readability** (0-1): Is the code clear and understandable?
    - Naming, comments, formatting, complexity
3. **Maintainability** (0-1): Can the code be easily changed?
    - Modularity, coupling, cohesion, testability
4. **Performance** (0-1): Is the code efficient?
    - Algorithm complexity, memory usage, bottlenecks
5. **Security** (0-1): Is the code safe from vulnerabilities?
    - Input validation, injection prevention, authentication

Overall Score: Weighted average (30% correctness, 25% readability, 25% maintainability, 10% performance, 10% security)

1.8 Refinement Loop

```
async function createWithRefinement(
  plan: Plan,
  threshold: number = 0.80
): Promise<Artifact> {
  let artifact = await generateInitial(plan);
  let quality = await measureQuality(artifact);
  let iterations = 0;
  const maxIterations = 3;

  while (
    quality.score < threshold &&
    iterations < maxIterations
  ) {
    // Identify specific issues
    const issues = quality.issues;

    // Refine based on issues
    artifact = await refine(artifact, issues);

    // Re-measure
    quality = await measureQuality(artifact);
    iterations++;
  }

  return {
    artifact,
    quality,
    iterations,
    success: quality.score >= threshold
  };
}
```

1.9 Template System

1.9.1 Available Templates

Classes: - `class-with-methods` - Basic class with methods - `service-class` - Service pattern with DI - `controller-class` - Controller with routes - `model-class` - Data model with validation

Functions: - `pure-function` - Pure functional approach - `async-function` - Async/await pattern - `error-handling-function` - Comprehensive error handling

Tests: - `unit-test-suite` - Jest unit tests - `integration-test` - Integration test patterns - `e2e-test` - End-to-end scenarios

1.9.2 Template Usage

```
// Use template
const code = await create({
  template: "service-class",
  context: {
    className: "PaymentService",
    methods: ["charge", "refund"],
    dependencies: ["StripeClient", "Logger"]
  }
});
```

1.10 Categorical Laws Verified

Free Monad laws:

1. **Return**: `Pure(a).bind(f) ≡ f(a)`
2. **Associativity**: `m.bind(f).bind(g) ≡ m.bind(x => f(x).bind(g))`

Coalgebra properties:

1. **Unfold/Fold**: `fold(unfold(pattern)) ≡ Some(pattern)`
2. **Uniqueness**: Unfold produces unique structure for given pattern

Status: ✅ Laws verified (in test suite)

1.11 Command-Line Usage

```
# Basic creation
cc2 create <plan.json>

# With quality threshold
cc2 create --threshold=0.85 <plan.json>

# Specific strategy
cc2 create --strategy=typescript <plan.json>

# Full chain
cc2 observe <state.json> | cc2 reason | cc2 create

# With template
cc2 create --template=service-class <context.json>
```

1.12 Performance

- **Cold Start**: <100ms (target after refactor)
- **Warm Execution**: <50ms per generation
- **Memory**: <300MB
- **Scalability**: O(n) for code size

Note: Currently ~150ms cold start due to size (6,064 lines), will improve to <100ms after Week 1 refactoring.

1.13 Current Limitations

- **Size**: 6,064 lines (refactoring to <2,500 in Week 1)
- **Languages**: TypeScript/JavaScript only (Python, Rust planned for Phase 2)
- **Templates**: ~20 templates (growing via template optimizer)
- **Quality**: Heuristic-based (will improve with ML in Phase 3)

1.14 Related Skills

- **cc2-observe**: Provides context for generation
- **cc2-reason**: Provides strategic plans
- **cc2-verify**: Validates generated code (Phase 2)
- **cc2-meta-orchestrator**: Orchestrates CREATE in workflows

---

1.15 References

- Implementation: `~/cc2.0/src/functions/create/`
- Tests: `~/cc2.0/src/functions/create/__tests__/`
- Documentation: `~/cc2.0/functions/create/FUNCTION.md`
- Integration: `~/cc2.0/functions/create/cc_integration.md`

1.16 Meta-Function: CREATE_SELF

CREATE can analyze and improve its own generation patterns:

```
// Meta-creation
const metaCreate = await createSelf(generationHistory);

// Returns insights about generation quality
{
  templateEffectiveness: {
    "service-class": { uses: 45, avgQuality: 0.87 },
    "controller-class": { uses: 32, avgQuality: 0.82 }
  },
  qualityTrends: "improving",
  antiPatternsReduced: ["magic-numbers", "god-object"],
  suggestions: [
    "Optimize service-class template (low quality variance)",
    "Add new template for repository pattern (frequent manual edits)"
  ]
}
```

See: `~/cc2.0/functions/create/modules/CREATE_SELF.md`

1.17 Integration Example: Full 3-Function Chain

```
// 1. OBSERVE system state
const observation = await observe({
  code: currentCode,
  metrics: performanceMetrics
});

// 2. REASON about improvements
const plan = await reason(observation);
// Strategy: "refactor", steps: [...]

// 3. CREATE implementation
const result = await create(plan);
// { artifact: {...}, quality: { score: 0.87 }, iterations: 2 }

// 4. Verify quality threshold met
if (result.quality.score >= 0.80) {
  console.log("✅ High-quality code generated");
  // Ready for deployment
} else {
  console.log("⚠️ Quality below threshold, manual review needed");
}
```

This demonstrates the complete categorical chain:

```
OBSERVE (Comonad) → REASON (Monad) → CREATE (Free Monad)
     ↓                  ↓                  ↓
  Context           Strategy           Artifact
```

Each step maintains type safety and categorical laws, ensuring correctness through composition.

1.18 Refactoring Status (Week 1 of Phase 1)

Current: 6,064 lines monolithic **Target**: <2,500 lines core + separate strategy modules **Timeline**: Week 1, Task 1.2 (20 hours) **Impact**: Improved maintainability, faster cold start, easier extension

Post-refactor structure:

```
create/
├── core/           (~800 lines)
├── strategies/     (~1,200 lines across 3 files)
├── generators/     (~1,200 lines across 3 files)
└── types.ts        (~200 lines)
```