```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 23
================================================================================
Generated: 2025-12-07T06:17:29.885346
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/utils/validation/schema_validator.py
================================================================================

"""
Schema validation for monolith initialization.

This module implements the Monolith Initialization Validator (MIV) that scans
and verifies the integrity of the global schema before runtime execution.
"""

import hashlib
import json
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

import jsonschema
from pydantic import BaseModel, ConfigDict, Field
from farfan_pipeline.core.calibration.decorators import calibrated_method


class SchemaInitializationError(Exception):
    """Raised when schema initialization validation fails."""
    pass

class MonolithIntegrityReport(BaseModel):
    """Report of monolith integrity validation."""

    model_config = ConfigDict(extra='allow')

    timestamp: str = Field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
    schema_version: str
    validation_passed: bool
    errors: list[str] = Field(default_factory=list)
    warnings: list[str] = Field(default_factory=list)
    schema_hash: str
    question_counts: dict[str, int]
    referential_integrity: dict[str, bool]

class MonolithSchemaValidator:
    """
    Monolith Initialization Validator (MIV).

    Bootstrapping process that scans and verifies the integrity of the
    global schema before runtime execution.
    """

    EXPECTED_SCHEMA_VERSION = "2.0.0"
    EXPECTED_MICRO_QUESTIONS = 300
    EXPECTED_MESO_QUESTIONS = 4
    EXPECTED_MACRO_QUESTIONS = 1
    EXPECTED_POLICY_AREAS = 10
    EXPECTED_DIMENSIONS = 6
    EXPECTED_CLUSTERS = 4

    def __init__(self, schema_path: str | None = None) -> None:
        """
        Initialize validator.

        Args:
            schema_path: Path to JSON schema file (optional)
        """
        self.schema_path = schema_path
        self.schema: dict[str, Any] | None = None
        self.errors: list[str] = []
        self.warnings: list[str] = []

        if schema_path:
            self._load_schema()

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._load_schema")
    def _load_schema(self, **kwargs: Any) -> None:
        """
        Load JSON schema from file.

        Handles file existence checks and JSON decoding errors with specific warnings.
        """
        if not self.schema_path:
            return

        schema_file = Path(self.schema_path)
        if not schema_file.exists():
            self.warnings.append(f"Schema file not found: {self.schema_path}")
            return

        try:
            with open(schema_file, mode='r', encoding='utf-8') as f:
                self.schema = json.load(f)
        except json.JSONDecodeError as e:
             self.warnings.append(f"Invalid JSON in schema file {self.schema_path}: {e}")
        except Exception as e:
            self.warnings.append(f"Unexpected error loading schema {self.schema_path}: {e}")

    def validate_monolith(
        self,
        monolith: dict[str, Any],
        strict: bool = True
    ) -> MonolithIntegrityReport:
        """
        Validate monolith structure and integrity.

        Args:
            monolith: Monolith configuration dictionary
            strict: If True, raises exception on validation failure

        Returns:
            MonolithIntegrityReport with validation results

        Raises:
            SchemaInitializationError: If validation fails and strict=True
        """
        self.errors = []
        self.warnings = []

        # 1. Validate structure
        self._validate_structure(monolith).value

        # 2. Validate schema version
        schema_version = self._validate_schema_version(monolith).value

        # 3. Validate question counts
        question_counts = self._validate_question_counts(monolith).value

        # 4. Validate referential integrity
        referential_integrity = self._validate_referential_integrity(monolith)

        # 5. Validate against JSON schema if available
        if self.schema:
            self._validate_against_schema(monolith).value

        # 6. Validate field coverage
        field_coverage = self._validate_field_coverage(monolith).value

        # 7. Validate semantic consistency
        semantic_consistency = self._validate_semantic_consistency(monolith).value

        # 8. Calculate schema hash
        schema_hash = self._calculate_schema_hash(monolith).value

        # Build report
        validation_passed = len(self.errors) == 0

        report = MonolithIntegrityReport(
            schema_version=schema_version,
            validation_passed=validation_passed,
            errors=self.errors,
            warnings=self.warnings,
            schema_hash=schema_hash,
            question_counts=question_counts,
            referential_integrity=referential_integrity
        )

        # Raise error if strict mode and validation failed
        if strict and not validation_passed:
            error_msg = "Schema initialization failed:\n" + "\n".join(
                f"  - {e}" for e in self.errors
            )
            raise SchemaInitializationError(error_msg)

        return report

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_structure")
    def _validate_structure(self, monolith: dict[str, Any], **kwargs: Any) -> None:
        """Validate top-level structure."""
        required_keys = ['schema_version', 'version', 'blocks', 'integrity']

        for key in required_keys:
            if key not in monolith:
                self.errors.append(f"Missing required top-level key: {key}")

        if 'blocks' in monolith:
            blocks = monolith['blocks']
            required_blocks = [
                'niveles_abstraccion',
                'micro_questions',
                'meso_questions',
                'macro_question',
                'scoring'
            ]

            for block in required_blocks:
                if block not in blocks:
                    self.errors.append(f"Missing required block: {block}")

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_schema_version")
    def _validate_schema_version(self, monolith: dict[str, Any], **kwargs: Any) -> str:
        """Validate schema version."""
        schema_version = monolith.get('schema_version', '')

        if not schema_version:
            self.errors.append("Missing schema_version")
            return ''

        # Allow any version but warn if not expected
        if schema_version != self.EXPECTED_SCHEMA_VERSION:
            self.warnings.append(
                f"Schema version {schema_version} differs from expected "
                f"{self.EXPECTED_SCHEMA_VERSION}"
            )

        return schema_version

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_question_counts")
    def _validate_question_counts(self, monolith: dict[str, Any], **kwargs: Any) -> dict[str, int]:
        """Validate question counts."""
        blocks = monolith.get('blocks', {})

        micro_count = len(blocks.get('micro_questions', []))
        meso_count = len(blocks.get('meso_questions', []))
        macro_exists = 1 if blocks.get('macro_question') else 0
        total_count = micro_count + meso_count + macro_exists

        # Validate counts
        if micro_count != self.EXPECTED_MICRO_QUESTIONS:
            self.errors.append(
                f"Expected {self.EXPECTED_MICRO_QUESTIONS} micro questions, "
                f"got {micro_count}"
            )

        if meso_count != self.EXPECTED_MESO_QUESTIONS:
            self.errors.append(
                f"Expected {self.EXPECTED_MESO_QUESTIONS} meso questions, "
                f"got {meso_count}"
            )

        if not macro_exists:
            self.errors.append("Missing macro question")

        expected_total = (
            self.EXPECTED_MICRO_QUESTIONS +
            self.EXPECTED_MESO_QUESTIONS +
            self.EXPECTED_MACRO_QUESTIONS
        )

        if total_count != expected_total:
            self.errors.append(
                f"Expected {expected_total} total questions, got {total_count}"
            )

        return {
            'micro': micro_count,
            'meso': meso_count,
            'macro': macro_exists,
            'total': total_count
        }

    def _validate_referential_integrity(
        self,
        monolith: dict[str, Any]
    ) -> dict[str, bool]:
        """
        Validate referential integrity.

        Ensures no dangling foreign keys or invalid cross-references.
        """
        results = {
            'policy_areas': True,
            'dimensions': True,
            'clusters': True,
            'micro_questions': True
        }

        blocks = monolith.get('blocks', {})
        niveles = blocks.get('niveles_abstraccion', {})

        # Get all valid IDs
        valid_policy_areas = {
            pa['policy_area_id']
            for pa in niveles.get('policy_areas', [])
        }

        valid_dimensions = {
            dim['dimension_id']
            for dim in niveles.get('dimensions', [])
        }

        valid_clusters = {
            cl['cluster_id']
            for cl in niveles.get('clusters', [])
        }

        # Validate cluster references to policy areas
        for cluster in niveles.get('clusters', []):
            cluster_id = cluster.get('cluster_id', 'UNKNOWN')
            for pa_id in cluster.get('policy_area_ids', []):
                if pa_id not in valid_policy_areas:
                    self.errors.append(
                        f"Cluster {cluster_id} references invalid policy area: {pa_id}"
                    )
                    results['clusters'] = False

        # Validate micro questions reference valid areas/dimensions
        for question in blocks.get('micro_questions', []):
            q_id = question.get('question_id', 'UNKNOWN')
            pa_id = question.get('policy_area_id')
            dim_id = question.get('dimension_id')

            if pa_id and pa_id not in valid_policy_areas:
                self.errors.append(
                    f"Question {q_id} references invalid policy area: {pa_id}"
                )
                results['micro_questions'] = False

            if dim_id and dim_id not in valid_dimensions:
                self.errors.append(
                    f"Question {q_id} references invalid dimension: {dim_id}"
                )
                results['micro_questions'] = False

        # Validate meso questions reference valid clusters
        for question in blocks.get('meso_questions', []):
            q_id = question.get('question_id', 'UNKNOWN')
            cl_id = question.get('cluster_id')

            if cl_id and cl_id not in valid_clusters:
                self.errors.append(
                    f"Meso question {q_id} references invalid cluster: {cl_id}"
                )

        return results

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_against_schema")
    def _validate_against_schema(self, monolith: dict[str, Any], **kwargs: Any) -> None:
        """Validate monolith against JSON schema."""
        if not self.schema:
            return

        try:
            jsonschema.validate(instance=monolith, schema=self.schema)
        except jsonschema.ValidationError as e:
            self.errors.append(f"Schema validation error: {e.message}")
        except Exception as e:
            self.warnings.append(f"Schema validation failed: {e}")

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_field_coverage")
    def _validate_field_coverage(self, monolith: dict[str, Any], **kwargs: Any) -> dict[str, float]:
        """
        Validate field coverage for micro-questions.

        Enforces >= 99% coverage for critical fields.
        """
        blocks = monolith.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])
        total_micro = len(micro_questions)

        if total_micro == 0:
            return {}

        critical_fields = [
            "question_id",
            "question_global",
            "base_slot",
            "dimension_id",
            "policy_area_id",
            "cluster_id",
            "scoring_modality",
            "scoring_definition_ref",
            "expected_elements",
            "method_sets",
            "failure_contract",
            "validations"
        ]

        coverage_stats = {}

        for field in critical_fields:
            present_count = 0
            for q in micro_questions:
                val = q.get(field)
                # Check for presence and non-emptiness (for lists/strings)
                if val is not None:
                    if isinstance(val, (list, dict, str)) and len(val) == 0:
                        pass # Empty container/string counts as missing for critical fields
                    else:
                        present_count += 1

            coverage = present_count / total_micro
            coverage_stats[field] = coverage

            if coverage < 0.99:
                self.errors.append(
                    f"Field coverage violation: '{field}' has {coverage:.2%} coverage, required >= 99%"
                )
            elif coverage < 1.0:
                self.warnings.append(
                    f"Field coverage warning: '{field}' has {coverage:.2%} coverage (should be 100%)"
                )

        return coverage_stats

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._validate_semantic_consistency")
    def _validate_semantic_consistency(self, monolith: dict[str, Any], **kwargs: Any) -> bool:
        """
        Validate semantic consistency of micro-questions.

        Checks:
        - question_global uniqueness and range (1-300)
        - base_slot format (D[1-6]-Q[1-5])
        - dimension_id format (DIM0[1-6])
        - policy_area_id format (PA0[1-9]|PA10)
        - cluster_id format (CL0[1-4])
        """
        blocks = monolith.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])

        seen_globals = set()
        all_valid = True

        import re
        base_slot_pattern = re.compile(r"^D[1-6]-Q[1-5]$")
        dim_pattern = re.compile(r"^DIM0[1-6]$")
        pa_pattern = re.compile(r"^PA(0[1-9]|10)$")
        cluster_pattern = re.compile(r"^CL0[1-4]$")

        for q in micro_questions:
            q_id = q.get("question_id", "UNKNOWN")
            q_global = q.get("question_global")

            # Check question_global
            if not isinstance(q_global, int) or not (1 <= q_global <= 300):
                self.errors.append(f"Question {q_id}: Invalid question_global {q_global} (must be 1-300)")
                all_valid = False
            elif q_global in seen_globals:
                self.errors.append(f"Question {q_id}: Duplicate question_global {q_global}")
                all_valid = False
            else:
                seen_globals.add(q_global)

            # Check patterns
            base_slot = q.get("base_slot")
            if not base_slot or not base_slot_pattern.match(base_slot):
                self.errors.append(f"Question {q_id}: Invalid base_slot '{base_slot}'")
                all_valid = False

            dim_id = q.get("dimension_id")
            if not dim_id or not dim_pattern.match(dim_id):
                self.errors.append(f"Question {q_id}: Invalid dimension_id '{dim_id}'")
                all_valid = False

            pa_id = q.get("policy_area_id")
            if not pa_id or not pa_pattern.match(pa_id):
                self.errors.append(f"Question {q_id}: Invalid policy_area_id '{pa_id}'")
                all_valid = False

            cluster_id = q.get("cluster_id")
            if not cluster_id or not cluster_pattern.match(cluster_id):
                self.errors.append(f"Question {q_id}: Invalid cluster_id '{cluster_id}'")
                all_valid = False

        # Check for gaps in question_global
        if len(seen_globals) == 300:
             expected_set = set(range(1, 301))
             if seen_globals != expected_set:
                 missing = expected_set - seen_globals
                 self.errors.append(f"Missing question_global values: {missing}")
                 all_valid = False

        return all_valid

    @calibrated_method("farfan_core.utils.validation.schema_validator.MonolithSchemaValidator._calculate_schema_hash")
    def _calculate_schema_hash(self, monolith: dict[str, Any], **kwargs: Any) -> str:
        """Calculate deterministic hash of monolith schema."""
        # Create canonical JSON representation
        canonical = json.dumps(monolith, sort_keys=True, ensure_ascii=True)

        # Calculate SHA-256 hash
        hash_obj = hashlib.sha256(canonical.encode('utf-8'))
        return hash_obj.hexdigest()

    def generate_validation_report(
        self,
        report: MonolithIntegrityReport,
        output_path: str
    ) -> None:
        """
        Generate and save validation report artifact.

        Args:
            report: Validation report
            output_path: Path to save report JSON
        """
        output_file = Path(output_path)
        output_file.parent.mkdir(parents=True, exist_ok=True)

        with open(output_file, 'w', encoding='utf-8') as f:
            json.dump(
                report.model_dump(),
                f,
                indent=2,
                ensure_ascii=False
            )

def validate_monolith_schema(
    monolith: dict[str, Any],
    schema_path: str | None = None,
    strict: bool = True
) -> MonolithIntegrityReport:
    """
    Convenience function to validate monolith schema.

    Args:
        monolith: Monolith configuration
        schema_path: Optional path to JSON schema
        strict: If True, raises exception on failure

    Returns:
        MonolithIntegrityReport

    Raises:
        SchemaInitializationError: If validation fails and strict=True
    """
    validator = MonolithSchemaValidator(schema_path=schema_path)
    return validator.validate_monolith(monolith, strict=strict)



================================================================================
FILE: src/farfan_pipeline/utils/validation_engine.py
================================================================================

#!/usr/bin/env python3
"""
Validation Engine - Centralized Rule-Based Validation
=====================================================

Provides a centralized validation engine with:
- Severity levels (ERROR/WARNING/INFO)
- Structured logging
- Context-aware precondition checking
- Integration with validation/predicates.py

Author: Integration Team - Agent 3
Version: 1.0.0
Python: 3.10+
"""

import logging
from dataclasses import dataclass, field
from datetime import datetime
from typing import Any

from farfan_pipeline.utils.validation.predicates import ValidationPredicates, ValidationResult
from farfan_pipeline.core.calibration.decorators import calibrated_method

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

```

```
@dataclass
class ValidationReport:
    """Complete validation report with all checks."""
    timestamp: str
    total_checks: int
    passed: int
    failed: int
    warnings: int
    results: list[ValidationResult] = field(default_factory=list)

    @calibrated_method("farfan_core.utils.validation_engine.ValidationReport.add_result")
    def add_result(self, result: ValidationResult) -> None:
        """Add a validation result to the report."""
        self.results.append(result)
        self.total_checks += 1

        if result.severity == "ERROR" and not result.is_valid:
            self.failed += 1
        elif result.severity == "WARNING":
            self.warnings += 1
        elif result.is_valid:
            self.passed += 1

    @calibrated_method("farfan_core.utils.validation_engine.ValidationReport.has_errors")
    def has_errors(self) -> bool:
        """Check if report contains any errors."""
        return self.failed > 0

    @calibrated_method("farfan_core.utils.validation_engine.ValidationReport.summary")
    def summary(self) -> str:
        """Generate summary string."""
        return (f"Validation Summary: {self.passed}/{self.total_checks} passed, "
                f"{self.failed} errors, {self.warnings} warnings")

class ValidationEngine:
    """
    Centralized validation engine for precondition checking.

    Integrates with validation/predicates.py to provide:
    - Precondition verification before execution steps
    - Structured validation reporting
    - Context-aware error messages
    - Severity-based logging (ERROR/WARNING/INFO)
    """

    def __init__(self, questionnaire_provider=None) -> None:
        """
        Initialize validation engine.

        Args:
            questionnaire_provider: QuestionnaireResourceProvider instance (injected via DI)
                                   If None, validation operations requiring questionnaire
                                   data will use ValidationPredicates without provider.

        ARCHITECTURAL NOTE: Direct cuestionario_data parameter REMOVED.
        Questionnaire access must go through QuestionnaireResourceProvider.
        """
        self.questionnaire_provider = questionnaire_provider
        self.predicates = ValidationPredicates()
        logger.info("ValidationEngine initialized")

    def validate_scoring_preconditions(
        self,
        question_spec: dict[str, Any],
        execution_results: dict[str, Any],
        plan_text: str
    ) -> ValidationResult:
        """
        Validate preconditions for scoring operations.

        Wraps ValidationPredicates.verify_scoring_preconditions with logging.

        Args:
            question_spec: Question specification from rubric
            execution_results: Results from execution pipeline
            plan_text: Full plan document text

        Returns:
            ValidationResult
        """
        logger.debug(f"Validating scoring preconditions for question: "
                    f"{question_spec.get('id', 'UNKNOWN')}")

        result = self.predicates.verify_scoring_preconditions(
            question_spec, execution_results, plan_text
        )

        self._log_result(result)
        return result

    def validate_expected_elements(
        self,
        question_spec: dict[str, Any]
    ) -> ValidationResult:
        """
        Validate expected_elements from questionnaire provider.

        Args:
            question_spec: Question specification

        Returns:
            ValidationResult
        """
        logger.debug(f"Validating expected_elements for question: "
                    f"{question_spec.get('id', 'UNKNOWN')}")

        # Get questionnaire data from provider if available
        questionnaire_data = {}
        if self.questionnaire_provider is not None:
            questionnaire_data = self.questionnaire_provider.get_data()

        result = self.predicates.verify_expected_elements(
                question_spec, questionnaire_data
            )

            self._log_result(result)
            return result

        def validate_execution_context(
            self,
            question_id: str,
            policy_area: str,
            dimension: str
        ) -> ValidationResult:
            """
            Validate execution context parameters.

            Args:
                question_id: Canonical question ID
                policy_area: Policy area (P1-P10)
                dimension: Dimension (D1-D6)

            Returns:
                ValidationResult
            """
            logger.debug(f"Validating execution context: {question_id}")

            result = self.predicates.verify_execution_context(
                question_id, policy_area, dimension
            )

            self._log_result(result)
            return result

        def validate_producer_availability(
            self,
            producer_name: str,
            producers_dict: dict[str, Any]
        ) -> ValidationResult:
            """
            Validate that producer is available and initialized.

            Args:
                producer_name: Name of the producer
                producers_dict: Dictionary of initialized producers

            Returns:
                ValidationResult
            """
            logger.debug(f"Validating producer availability: {producer_name}")

            result = self.predicates.verify_producer_availability(
                producer_name, producers_dict
            )

            self._log_result(result)
            return result

```

```
    def validate_all_preconditions(
        self,
        question_spec: dict[str, Any],
        execution_results: dict[str, Any],
        plan_text: str,
        producers_dict: dict[str, Any]
    ) -> ValidationReport:
        """
        Run all validation checks for a question execution.

        Args:
            question_spec: Question specification
            execution_results: Execution results
            plan_text: Plan document text
            producers_dict: Initialized producers

        Returns:
            ValidationReport with all checks
        """
        report = ValidationReport(
            timestamp=datetime.now().isoformat(),
            total_checks=0,
            passed=0,
            failed=0,
            warnings=0
        )

        logger.info("=" * 80)
        logger.info(f"Running validation checks for: {question_spec.get('id', 'UNKNOWN')}")
        logger.info("=" * 80)

        # Check 1: Execution context
        question_id = question_spec.get("id", "")
        policy_area = question_spec.get("policy_area", "")
        dimension = question_spec.get("dimension", "")

        result = self.validate_execution_context(question_id, policy_area, dimension)
        report.add_result(result)

        # Check 2: Expected elements
        result = self.validate_expected_elements(question_spec)
        report.add_result(result)

        # Check 3: Scoring preconditions
        result = self.validate_scoring_preconditions(
            question_spec, execution_results, plan_text
        )
        report.add_result(result)

        # Check 4: Producer availability (if specified)
        evidence_sources = question_spec.get("evidence_sources", {})
        orchestrator_key = evidence_sources.get("orchestrator_key", "")

        if orchestrator_key:
            # Handle both string and list formats
            if isinstance(orchestrator_key, str):
                result = self.validate_producer_availability(
                    orchestrator_key, producers_dict
                )
                report.add_result(result)
            elif isinstance(orchestrator_key, list):
                for producer in orchestrator_key:
                    result = self.validate_producer_availability(
                        producer, producers_dict
                    )
                    report.add_result(result)

        logger.info("=" * 80)
        logger.info(report.summary())
        logger.info("=" * 80)

        return report

    @calibrated_method("farfan_core.utils.validation_engine.ValidationEngine._log_result")
    def _log_result(self, result: ValidationResult) -> None:
        """Log validation result with appropriate severity."""
        if result.severity == "ERROR":
            if result.is_valid:
                logger.debug(f"â\234\223 {result.message}")
            else:
                logger.error(f"â\234\227 {result.message}")
        elif result.severity == "WARNING":
            logger.warning(f"â\232  {result.message}")
        else:
            logger.debug(f"â\204¹ {result.message}")

    def create_validation_report(
        self,
        results: list[ValidationResult]
    ) -> ValidationReport:
        """
        Create a validation report from a list of results.

        Args:
            results: List of ValidationResult objects

        Returns:
            ValidationReport
        """
        report = ValidationReport(
            timestamp=datetime.now().isoformat(),
            total_checks=0,
            passed=0,
            failed=0,
            warnings=0
        )

        for result in results:
            report.add_result(result)

        return report

```

```


================================================================================
FILE: system/artifacts/inventory/method_inventory_scanner.py
================================================================================

#!/usr/bin/env python3
"""
FARFAN Method Inventory Scanner - PRODUCTION GRADE
Scans repository for all analytical methods with AST-based extraction.

REQUIREMENTS:
- Coverage: â\211¥95% of executor methods
- Accuracy: 0 phantom methods
- Determinism: Hash(run1) == Hash(run2)
- Performance: < 10 seconds
"""
import ast
import hashlib
import json
import sys
import time
from dataclasses import dataclass, asdict
from datetime import datetime, timezone
from pathlib import Path
from typing import Dict, List, Set, Optional, Any

# MANDATORY SCAN PATHS - Verified to exist
MANDATORY_SCAN_PATHS = [
    "farfan_core/farfan_core/core",
    "farfan_core/farfan_core/processing",
    "farfan_core/farfan_core/analysis",
    "farfan_core/farfan_core/orchestrator",
    "farfan_core/farfan_core/utils",
]

EXCLUDE_PATTERNS = [
    "__pycache__",
    ".git",
    ".venv",
    "venv",
    "node_modules",
    "build",
    "dist",
    ".pytest_cache",
    ".mypy_cache",
]


@dataclass
class MethodSignature:
    """Method signature information"""
    args: List[str]
    kwargs: List[str]
    returns: Optional[str]
    is_async: bool
    accepts_executor_config: bool
    decorators: List[str]


@dataclass
class GovernanceFlags:
    """Governance and compliance flags"""
    uses_yaml: bool
    has_hardcoded_calibration: bool
    has_hardcoded_timeout: bool
    suspicious_magic_numbers: List[float]
    is_executor_class: bool


@dataclass
class LocationInfo:
    """Source location information"""
    file_path: str
    line_start: int
    line_end: int


@dataclass
class MethodDescriptor:
    """Complete method descriptor"""
    method_id: str
    role: str
    aggregation_level: str
    module: str
    class_name: Optional[str]
    method_name: str
    signature: MethodSignature
    governance_flags: GovernanceFlags
    location: LocationInfo
    ast_hash: str
    docstring: Optional[str]
    complexity: int
    dependencies: List[str]


class MethodInventoryScanner:
    """Production-grade method inventory scanner with full validation"""

    def __init__(self, repo_root: Path):
        self.repo_root = repo_root.resolve()
        self.inventory: Dict[str, MethodDescriptor] = {}
        self.failures: List[Dict[str, str]] = []
        self.files_scanned = 0
        self.start_time = time.time()

    def scan(self) -> Dict[str, Any]:
        """Execute full scan with validation"""
        print(f"ð\237\224\215 Scanning repository: {self.repo_root}")

        # STEP 1: Verify all mandatory paths exist
        self._verify_paths()

        # STEP 2: Load questionnaire_monolith.json for cross-reference (if exists)
        monolith_methods = self._load_monolith_methods()

        # STEP 3: Scan all Python files
        python_files = self._find_python_files()
        print(f"ð\237\223\201 Found {len(python_files)} Python files to scan")

        # STEP 4: Parse each file and extract methods
        for py_file in python_files:
            self._scan_file(py_file)

        # STEP 5: Cross-reference with monolith
        if monolith_methods:
            self._cross_reference_monolith(monolith_methods)

        # STEP 6: Generate manifest
        return self._generate_manifest()

    def _verify_paths(self):
        """Verify all mandatory scan paths exist"""
        print("â\234\223 Verifying mandatory paths...")
        for path_str in MANDATORY_SCAN_PATHS:
            full_path = self.repo_root / path_str
            if not full_path.exists():
                error = f"Missing mandatory path: {path_str}"
                self.failures.append({
                    "type": "missing_path",
                    "path": path_str,
                    "error": error
                })
                print(f"  â\235\214 {error}")
            else:
                print(f"  â\234\223 {path_str}")

    def _load_monolith_methods(self) -> Set[str]:
        """Load method references from questionnaire_monolith.json"""
        monolith_path = self.repo_root / "system/config/questionnaire/questionnaire_monolith.json"

        if not monolith_path.exists():
            print(f"â\232 ï¸\217  Monolith not found (optional): {monolith_path}")
            return set()

        try:
            with open(monolith_path) as f:
                monolith = json.load(f)

            # Extract all method references from method_sets
            methods = set()
            for question in monolith.get("questions", []):
                for method_set in question.get("method_sets", {}).values():
                    methods.update(method_set.get("methods", []))

            print(f"â\234\223 Loaded {len(methods)} method references from monolith")
            return methods
        except Exception as e:
            print(f"â\232 ï¸\217  Failed to load monolith: {e}")
            return set()

    def _find_python_files(self) -> List[Path]:
        """Find all Python files in mandatory scan paths"""
        python_files = []

        for path_str in MANDATORY_SCAN_PATHS:
            scan_path = self.repo_root / path_str
            if not scan_path.exists():
                continue

            for py_file in scan_path.rglob("*.py"):
                # Skip excluded patterns
                if any(pattern in str(py_file) for pattern in EXCLUDE_PATTERNS):
                    continue
                python_files.append(py_file)

        return sorted(python_files)

    def _scan_file(self, file_path: Path):
        """Scan a single Python file for methods"""
        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                source = f.read()

            tree = ast.parse(source, filename=str(file_path))
            self.files_scanned += 1

            # Extract module path
            rel_path = file_path.relative_to(self.repo_root)
            module = self._path_to_module(rel_path)

            # Visit all nodes
            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef) or isinstance(node, ast.AsyncFunctionDef):
                    self._extract_method(node, module, str(rel_path), None)
                elif isinstance(node, ast.ClassDef):
                    for item in node.body:
                        if isinstance(item, (ast.FunctionDef, ast.AsyncFunctionDef)):
                            self._extract_method(item, module, str(rel_path), node.name)

        except SyntaxError as e:
            self.failures.append({
                "type": "syntax_error",
                "file": str(file_path.relative_to(self.repo_root)),
                "error": str(e)
            })
        except Exception as e:
            self.failures.append({
                "type": "parse_error",
                "file": str(file_path.relative_to(self.repo_root)),
                "error": str(e)
            })

    def _path_to_module(self, path: Path) -> str:
        """Convert file path to module name"""
        parts = list(path.parts)

        # Remove .py extension
        if parts[-1].endswith('.py'):
            parts[-1] = parts[-1][:-3]

        # Remove __init__
        if parts[-1] == '__init__':
            parts = parts[:-1]

        return '.'.join(parts)

    def _extract_method(self, node: ast.FunctionDef, module: str, file_path: str,
                       class_name: Optional[str]):
        """Extract method descriptor from AST node"""
        method_name = node.name

        # Skip private methods starting with _
        if method_name.startswith('_') and method_name != '__init__':
            return

        # Build method_id
        if class_name:
            method_id = f"{class_name}.{method_name}"
        else:
            method_id = method_name

        # Extract signature
        signature = self._extract_signature(node)

        # Extract governance flags
        governance = self._extract_governance_flags(node)

        # Compute AST hash
        ast_hash = self._compute_ast_hash(node)

        # Extract docstring
        docstring = ast.get_docstring(node)

        # Compute cyclomatic complexity
        complexity = self._compute_complexity(node)

        # Extract dependencies (method calls)
        dependencies = self._extract_dependencies(node)

        # Infer role and aggregation level
        role, agg_level = self._infer_role(method_id, class_name, file_path)

        # Create descriptor
        descriptor = MethodDescriptor(
            method_id=method_id,
            role=role,
            aggregation_level=agg_level,
            module=module,
            class_name=class_name,
            method_name=method_name,
            signature=signature,
            governance_flags=governance,
            location=LocationInfo(
                file_path=file_path,
                line_start=node.lineno,
                line_end=node.end_lineno or node.lineno
            ),
            ast_hash=ast_hash,
            docstring=docstring,
            complexity=complexity,
            dependencies=dependencies
        )

        self.inventory[method_id] = descriptor

    def _extract_signature(self, node: ast.FunctionDef) -> MethodSignature:
        """Extract method signature"""
        args = [arg.arg for arg in node.args.args]
        kwargs = [arg.arg for arg in node.args.kwonlyargs]

        # Extract return type
        returns = None
        if node.returns:
            returns = ast.unparse(node.returns)

        # Check if async
        is_async = isinstance(node, ast.AsyncFunctionDef)

        # Check if accepts executor_config
        accepts_config = 'config' in args or 'executor_config' in args

        # Extract decorators
        decorators = [ast.unparse(dec) for dec in node.decorator_list]

        return MethodSignature(
            args=args,
            kwargs=kwargs,
            returns=returns,
            is_async=is_async,
            accepts_executor_config=accepts_config,
            decorators=decorators
        )

    def _extract_governance_flags(self, node: ast.FunctionDef) -> GovernanceFlags:
        """Extract governance and compliance flags"""
        source = ast.unparse(node)

        # Check for YAML usage
        uses_yaml = 'yaml' in source.lower() or '.yml' in source.lower()

        # Check for hardcoded calibration values
        has_hardcoded_cal = any(keyword in source for keyword in [
            'b_theory', 'b_impl', 'b_deploy', 'calibration_score'
        ])

        # Check for hardcoded timeouts
        has_hardcoded_timeout = 'timeout' in source and any(
            f'timeout={val}' in source or f'timeout = {val}' in source
            for val in range(1, 100)
        )

        # Detect suspicious magic numbers
        magic_numbers = []
        for n in ast.walk(node):
            if isinstance(n, ast.Constant) and isinstance(n.value, (int, float)):
                if n.value not in [0, 1, -1, 2, 10, 100, 1000]:
                    magic_numbers.append(float(n.value))

        # Check if executor class
        is_executor = 'Executor' in (node.name if hasattr(node, 'name') else '')

        return GovernanceFlags(
            uses_yaml=uses_yaml,
            has_hardcoded_calibration=has_hardcoded_cal,
            has_hardcoded_timeout=has_hardcoded_timeout,
            suspicious_magic_numbers=magic_numbers[:5],  # Limit to 5
            is_executor_class=is_executor
        )

    def _compute_ast_hash(self, node: ast.FunctionDef) -> str:
        """Compute deterministic SHA256 hash of normalized AST"""
        # Normalize AST by unparsing and re-parsing to remove formatting
        normalized = ast.unparse(node)
        return hashlib.sha256(normalized.encode('utf-8')).hexdigest()

    def _compute_complexity(self, node: ast.FunctionDef) -> int:
        """Compute cyclomatic complexity"""
        complexity = 1  # Base complexity

        for n in ast.walk(node):
            # Count decision points
            if isinstance(n, (ast.If, ast.While, ast.For, ast.ExceptHandler)):
                complexity += 1
            elif isinstance(n, ast.BoolOp):
                complexity += len(n.values) - 1

        return complexity

    def _extract_dependencies(self, node: ast.FunctionDef) -> List[str]:
        """Extract method call dependencies"""
        dependencies = set()

        for n in ast.walk(node):
            if isinstance(n, ast.Call):
                if isinstance(n.func, ast.Attribute):
                    # Method call: obj.method()
                    dependencies.add(n.func.attr)
                elif isinstance(n.func, ast.Name):
                    # Function call: function()
                    dependencies.add(n.func.id)

        return sorted(list(dependencies))[:20]  # Limit to 20

    def _infer_role(self, method_id: str, class_name: Optional[str],
                    file_path: str) -> tuple[str, str]:
        """Infer method role and aggregation level"""
        role = "UNKNOWN"
        agg_level = "UNKNOWN"

        # Infer from class name
        if class_name:
            if 'Executor' in class_name:
                role = "EXECUTOR"
                agg_level = "LEVEL_3"
            elif 'Processor' in class_name:
                role = "PROCESSOR"
                agg_level = "LEVEL_4"
            elif 'Analyzer' in class_name:
                role = "ANALYZER"
                agg_level = "LEVEL_4"

        # Infer from file path
        if 'orchestrator' in file_path:
            role = "ENGINE"
            agg_level = "LEVEL_0"
        elif 'processing' in file_path:
            role = "PROCESSOR"
        elif 'analysis' in file_path:
            role = "ANALYZER"

        return role, agg_level

    def _cross_reference_monolith(self, monolith_methods: Set[str]):
        """Cross-reference inventory with monolith"""
        inventory_methods = set(self.inventory.keys())

        # Find phantom methods (in monolith but not in inventory)
        phantom = monolith_methods - inventory_methods
        if phantom:
            print(f"â\232 ï¸\217  Found {len(phantom)} phantom methods in monolith:")
            for method in sorted(phantom)[:10]:
                print(f"    - {method}")

        # Find orphan methods (in inventory but not in monolith)
        orphan = inventory_methods - monolith_methods
        print(f"â\204¹ï¸\217  Found {len(orphan)} methods not referenced in monolith")

    def _generate_manifest(self) -> Dict[str, Any]:
        """Generate final manifest"""
        elapsed = time.time() - self.start_time

        manifest = {
            "metadata": {
                "scan_timestamp": datetime.now(timezone.utc).isoformat(),
                "repository": "PEROPOROBTANTE/F.A.R.F.A.N-MECHANISTIC_POLICY_PIPELINE_FINAL",
                "total_files_scanned": self.files_scanned,
                "total_methods_found": len(self.inventory),
                "ast_parse_failures": self.failures,
                "scan_duration_seconds": round(elapsed, 2)
            },
            "methods": {
                method_id: asdict(descriptor)
                for method_id, descriptor in sorted(self.inventory.items())
            },
            "_version": "2.0.0",
            "_comment": "Method inventory with AST hashing and cross-references"
        }

        return manifest

    def compute_checksum(self) -> str:
        """Compute deterministic checksum of inventory"""
        # Sort keys for determinism
        sorted_methods = sorted(self.inventory.items())

        # Compute hash of all AST hashes
        combined = ''.join(desc.ast_hash for _, desc in sorted_methods)
        return hashlib.sha256(combined.encode('utf-8')).hexdigest()


def main():
    """Main entry point"""
    import argparse

    parser = argparse.ArgumentParser(description="FARFAN Method Inventory Scanner")
    parser.add_argument('--repo-root', type=Path, default=Path('.'),
                        help='Repository root path')
    parser.add_argument('--output', type=Path,
                        default=Path('method_inventory_verified.json'),
                        help='Output JSON file')
    parser.add_argument('--checksum', action='store_true',
                        help='Output only checksum')

    args = parser.parse_args()

    # Create scanner
    scanner = MethodInventoryScanner(args.repo_root)

    # Execute scan
    manifest = scanner.scan()

    if args.checksum:
        # Output checksum only
        checksum = scanner.compute_checksum()
        print(checksum)
    else:
        # Save manifest
        with open(args.output, 'w') as f:
            json.dump(manifest, f, indent=2)

        print(f"\nâ\234\205 Scan complete!")
        print(f"   Files scanned: {manifest['metadata']['total_files_scanned']}")
        print(f"   Methods found: {manifest['metadata']['total_methods_found']}")
        print(f"   Parse failures: {len(manifest['metadata']['ast_parse_failures'])}")
        print(f"   Duration: {manifest['metadata']['scan_duration_seconds']}s")
        print(f"   Output: {args.output}")

        # Compute and display checksum
        checksum = scanner.compute_checksum()
        print(f"   Checksum: {checksum}")

        return 0 if len(scanner.failures) == 0 else 1


if __name__ == '__main__':
    sys.exit(main())



================================================================================
FILE: system/config/config_cli.py
================================================================================

#!/usr/bin/env python3
"""Command-line interface for configuration management.

Usage examples:
    # Save a config file
    python config_cli.py save calibration/model.json --data '{"model": "test"}'

    # Load a config file
    python config_cli.py load calibration/model.json

    # Verify file hash
    python config_cli.py verify calibration/model.json

    # List backups
    python config_cli.py backups calibration/model.json

    # Restore from backup
    python config_cli.py restore .backup/20231201_143000_calibration_model.json

    # Show registry
    python config_cli.py registry

    # Rebuild registry
    python config_cli.py rebuild
"""

import argparse
import json
import sys
from pathlib import Path

from config_manager import ConfigManager


def main() -> int:
    """Main CLI entry point."""
    parser = argparse.ArgumentParser(
        description="Configuration management with hash tracking and backups"
    )
    subparsers = parser.add_subparsers(dest="command", help="Command to execute")

    save_parser = subparsers.add_parser("save", help="Save configuration file")
    save_parser.add_argument("path", help="Relative path to config file")
    save_parser.add_argument("--data", help="JSON data to save", required=True)
    save_parser.add_argument(
        "--no-backup", action="store_true", help="Skip backup creation"
    )

    load_parser = subparsers.add_parser("load", help="Load configuration file")
    load_parser.add_argument("path", help="Relative path to config file")
    load_parser.add_argument("--json", action="store_true", help="Parse as JSON")

    verify_parser = subparsers.add_parser("verify", help="Verify file hash")
    verify_parser.add_argument("path", help="Relative path to config file")

    info_parser = subparsers.add_parser("info", help="Show file information")
    info_parser.add_argument("path", help="Relative path to config file")

    backups_parser = subparsers.add_parser("backups", help="List backups")
    backups_parser.add_argument("path", nargs="?", help="Optional path filter")

    restore_parser = subparsers.add_parser("restore", help="Restore from backup")
    restore_parser.add_argument("backup", help="Backup file name or path")
    restore_parser.add_argument(
        "--no-backup", action="store_true", help="Skip backup of current version"
    )

    subparsers.add_parser("registry", help="Show hash registry")

    subparsers.add_parser("rebuild", help="Rebuild hash registry")

    args = parser.parse_args()

    if args.command is None:
        parser.print_help()
        return 1

    manager = ConfigManager()

    try:
        if args.command == "save":
            try:
                data = json.loads(args.data)
                file_path = manager.save_config_json(
                    args.path, data, create_backup=not args.no_backup
                )
            except json.JSONDecodeError:
                file_path = manager.save_config(
                    args.path, args.data, create_backup=not args.no_backup
                )
            print(f"Saved: {file_path}")
            info = manager.get_file_info(args.path)
            print(f"Hash: {info['hash']}")

        elif args.command == "load":
            if args.json:
                data = manager.load_config_json(args.path)
                print(json.dumps(data, indent=2))
            else:
                content = manager.load_config(args.path)
                print(content)

        elif args.command == "verify":
            is_valid = manager.verify_hash(args.path)
            if is_valid:
                print(f"â\234\223 Hash verification passed: {args.path}")
                return 0
            else:
                print(f"â\234\227 Hash verification failed: {args.path}", file=sys.stderr)
                return 1

        elif args.command == "info":
            info = manager.get_file_info(args.path)
            if info is None:
                print(f"File not in registry: {args.path}", file=sys.stderr)
                return 1
            print(json.dumps(info, indent=2))

        elif args.command == "backups":
            backups = manager.list_backups(args.path)
            if not backups:
                print("No backups found")
            else:
                print(f"Found {len(backups)} backup(s):")
                for backup in backups:
                    print(f"  {backup.name}")

        elif args.command == "restore":
            backup_path = Path(args.backup)
            if not backup_path.is_absolute():
                backup_path = manager.backup_dir / args.backup

            restored_path = manager.restore_backup(
                backup_path, create_backup=not args.no_backup
            )
            print(f"Restored: {restored_path}")
            info = manager.get_file_info(
                str(restored_path.relative_to(manager.config_root))
            )
            print(f"Hash: {info['hash']}")

        elif args.command == "registry":
            registry = manager.get_registry()
            print(json.dumps(registry, indent=2))

        elif args.command == "rebuild":
            print("Rebuilding hash registry...")
            registry = manager.rebuild_registry()
            print(f"Registered {len(registry)} file(s)")
            for path in sorted(registry.keys()):
                print(f"  {path}")

    except Exception as e:
        print(f"Error: {e}", file=sys.stderr)
        return 1

    return 0


if __name__ == "__main__":
    sys.exit(main())



================================================================================
FILE: system/config/config_manager.py
================================================================================

"""Configuration management system with SHA256 hash registry and timestamped backups.

Provides atomic config file operations with:
- Automatic SHA256 hash computation and registry tracking
- Timestamped backups before modifications (YYYYMMDD_HHMMSS format)
- JSON-based hash registry (config_hash_registry.json)
"""

import hashlib
import json
import shutil
from datetime import datetime
from pathlib import Path
from typing import Any

BACKUP_FILENAME_MIN_PARTS = 4


class ConfigManager:
    """Manages configuration files with hash tracking and automatic backups."""

    def __init__(self, config_root: Path | None = None) -> None:
        """Initialize config manager.

        Args:
            config_root: Root directory for config files. Defaults to system/config.
        """
        if config_root is None:
            config_root = Path(__file__).parent
        self.config_root = Path(config_root)
        self.backup_dir = self.config_root / ".backup"
        self.registry_file = self.config_root / "config_hash_registry.json"
        self._ensure_directories()

    def _ensure_directories(self) -> None:
        """Ensure required directories exist."""
        self.backup_dir.mkdir(parents=True, exist_ok=True)
        for subdir in ["calibration", "questionnaire", "environments"]:
            (self.config_root / subdir).mkdir(parents=True, exist_ok=True)

    def _compute_sha256(self, file_path: Path) -> str:
        """Compute SHA256 hash of file.

        Args:
            file_path: Path to file to hash

        Returns:
            Hexadecimal SHA256 hash string
        """
        sha256_hash = hashlib.sha256()
        with open(file_path, "rb") as f:
            for byte_block in iter(lambda: f.read(4096), b""):
                sha256_hash.update(byte_block)
        return sha256_hash.hexdigest()

    def _load_registry(self) -> dict[str, Any]:  # type: ignore[misc]
        """Load hash registry from disk.

        Returns:
            Registry dictionary with file paths as keys
        """
        if not self.registry_file.exists():
            return {}
        with open(self.registry_file) as f:
            return json.load(f)  # type: ignore[no-any-return]

    def _save_registry(self, registry: dict[str, Any]) -> None:  # type: ignore[misc]
        """Save hash registry to disk.

        Args:
            registry: Registry dictionary to save
        """
        with open(self.registry_file, "w") as f:
            json.dump(registry, f, indent=2, sort_keys=True)

    def _create_backup(self, file_path: Path) -> Path | None:
        """Create timestamped backup of config file.

        Args:
            file_path: Path to file to backup

        Returns:
            Path to backup file, or None if source doesn't exist
        """
        if not file_path.exists():
            return None

        now = datetime.now()
        timestamp = now.strftime("%Y%m%d_%H%M%S")
        microseconds = now.strftime("%f")
        relative_path = file_path.relative_to(self.config_root)
        backup_name = (
            f"{timestamp}_{microseconds}_{relative_path.as_posix().replace('/', '_')}"
        )
        backup_path = self.backup_dir / backup_name

        shutil.copy2(file_path, backup_path)
        return backup_path

    def _update_registry(self, file_path: Path) -> None:
        """Update hash registry for a file.

        Args:
            file_path: Path to file to register
        """
        if not file_path.exists():
            return

        registry = self._load_registry()
        relative_path = str(file_path.relative_to(self.config_root))
        hash_value = self._compute_sha256(file_path)

        registry[relative_path] = {
            "hash": hash_value,
            "last_modified": datetime.now().isoformat(),
            "size_bytes": file_path.stat().st_size,
        }

        self._save_registry(registry)

    def save_config(
        self, relative_path: str, content: str, create_backup: bool = True
    ) -> Path:
        """Save configuration file with automatic backup and hash registry update.

        Args:
            relative_path: Path relative to config root (e.g., 'calibration/model.json')
            content: Content to write to file
            create_backup: Whether to create backup before modification

        Returns:
            Path to saved config file
        """
        file_path = self.config_root / relative_path

        if create_backup and file_path.exists():
            self._create_backup(file_path)

        file_path.parent.mkdir(parents=True, exist_ok=True)
        file_path.write_text(content, encoding="utf-8")

        self._update_registry(file_path)

        return file_path

    def save_config_json(  # type: ignore[misc]
        self, relative_path: str, data: Any, create_backup: bool = True
    ) -> Path:
        """Save JSON configuration file with automatic backup and hash registry update.

        Args:
            relative_path: Path relative to config root (e.g., ’calibration/model.json’)
            data: Data to serialize to JSON
            create_backup: Whether to create backup before modification

        Returns:
            Path to saved config file
        """
        content = json.dumps(data, indent=2, sort_keys=True)
        return self.save_config(relative_path, content, create_backup=create_backup)

    def load_config(self, relative_path: str) -> str:
        """Load configuration file content.

        Args:
            relative_path: Path relative to config root

        Returns:
            File content as string

        Raises:
            FileNotFoundError: If config file doesn’t exist
        """
        file_path = self.config_root / relative_path
        return file_path.read_text(encoding="utf-8")

    def load_config_json(self, relative_path: str) -> Any:  # type: ignore[misc]
        """Load JSON configuration file.

        Args:
            relative_path: Path relative to config root

        Returns:
            Parsed JSON data

        Raises:
            FileNotFoundError: If config file doesn’t exist
        """
        content = self.load_config(relative_path)
        return json.loads(content)

    def verify_hash(self, relative_path: str) -> bool:
        """Verify file hash matches registry.

        Args:
            relative_path: Path relative to config root

        Returns:
            True if hash matches registry, False otherwise
        """
        file_path = self.config_root / relative_path
        if not file_path.exists():
            return False

        registry = self._load_registry()
        if relative_path not in registry:
            return False

        current_hash = self._compute_sha256(file_path)
        return current_hash == registry[relative_path]["hash"]  # type: ignore[no-any-return]

    def get_file_info(self, relative_path: str) -> dict[str, Any] | None:  # type: ignore[misc]
        """Get registry information for a file.

        Args:
            relative_path: Path relative to config root

        Returns:
            Registry entry dict or None if not registered
        """
        registry = self._load_registry()
        return registry.get(relative_path)

    def list_backups(self, relative_path: str | None = None) -> list[Path]:
        """List backup files.

        Args:
            relative_path: Optional filter for specific config file

        Returns:
            List of backup file paths, sorted by timestamp (newest first)
        """
        backups = []
        pattern = (
            "*" if relative_path is None else f"*_{relative_path.replace('/', '_')}"
        )

        for backup_file in self.backup_dir.glob(pattern):
            if backup_file.is_file():
                backups.append(backup_file)

        return sorted(backups, reverse=True)

    def restore_backup(self, backup_path: Path, create_backup: bool = True) -> Path:
        """Restore configuration from backup.

        Args:
            backup_path: Path to backup file
            create_backup: Whether to backup current file before restore

        Returns:
            Path to restored config file

        Raises:
            ValueError: If backup path format is invalid
        """
        if not backup_path.exists():
            raise FileNotFoundError(f"Backup file not found: {backup_path}")

        backup_name = backup_path.name
        if "_" not in backup_name:
            raise ValueError(f"Invalid backup filename format: {backup_name}")

        parts = backup_name.split("_", 3)
        if len(parts) < BACKUP_FILENAME_MIN_PARTS:
            raise ValueError(f"Invalid backup filename format: {backup_name}")

        encoded_path = parts[3]
        registry = self._load_registry()

        relative_path = None
        for path in registry:
            if path.replace("/", "_") == encoded_path:
                relative_path = path
                break

        if relative_path is None:
            relative_path = encoded_path.replace("_", "/")

        file_path = self.config_root / relative_path

        if create_backup and file_path.exists():
            self._create_backup(file_path)

        file_path.parent.mkdir(parents=True, exist_ok=True)
        shutil.copy2(backup_path, file_path)

        self._update_registry(file_path)

        return file_path

    def get_registry(self) -> dict[str, Any]:  # type: ignore[misc]
        """Get complete hash registry.

        Returns:
            Registry dictionary
        """
        return self._load_registry()

    def rebuild_registry(self) -> dict[str, Any]:  # type: ignore[misc]
        """Rebuild hash registry by scanning all config files.

        Returns:
            Updated registry dictionary
        """
        registry = {}
        for pattern in ["**/*.json", "**/*.yaml", "**/*.yml", "**/*.toml"]:
            for file_path in self.config_root.glob(pattern):
                if file_path.is_file() and ".backup" not in file_path.parts:
                    relative_path = str(file_path.relative_to(self.config_root))
                    hash_value = self._compute_sha256(file_path)
                    registry[relative_path] = {
                        "hash": hash_value,
                        "last_modified": datetime.fromtimestamp(
                            file_path.stat().st_mtime
                        ).isoformat(),
                    "size_bytes": file_path.stat().st_size,
                }

        self._save_registry(registry)
        return registry



================================================================================
FILE: system/config/example_usage.py
================================================================================

"""Example usage of the configuration management system."""


from config_manager import ConfigManager


def main() -> None:
    """Demonstrate config manager usage."""
    manager = ConfigManager()

    calibration_config = {
        "model": "gpt-4",
        "temperature": 0.7,
        "max_tokens": 2000,
        "thresholds": {"low": 0.3, "medium": 0.6, "high": 0.9},
    }

    print("=== Saving Configuration ===")
    config_path = manager.save_config_json(
        "calibration/model_config.json", calibration_config
    )
    print(f"Saved to: {config_path}")

    info = manager.get_file_info("calibration/model_config.json")
    print(f"Hash: {info['hash'][:16]}...")
    print(f"Size: {info['size_bytes']} bytes")

    print("\n=== Modifying Configuration ===")
    calibration_config["temperature"] = 0.8
    manager.save_config_json("calibration/model_config.json", calibration_config)
    print("Configuration updated (backup created automatically)")

    print("\n=== Listing Backups ===")
    backups = manager.list_backups("calibration/model_config.json")
    print(f"Found {len(backups)} backup(s):")
    for backup in backups:
        print(f"  - {backup.name}")

    print("\n=== Hash Verification ===")
    is_valid = manager.verify_hash("calibration/model_config.json")
    print(f"Hash verification: {'â\234\223 PASS' if is_valid else 'â\234\227 FAIL'}")

    print("\n=== Loading Configuration ===")
    loaded_config = manager.load_config_json("calibration/model_config.json")
    print(f"Temperature: {loaded_config['temperature']}")
    print(f"Thresholds: {loaded_config['thresholds']}")

    print("\n=== Registry Summary ===")
    registry = manager.get_registry()
    print(f"Total registered files: {len(registry)}")
    for path in registry:
        print(f"  - {path}")

    print("\n=== Environment Config Example ===")
    env_config = {
        "environment": "production",
        "database_url": "postgresql://prod-server/db",
        "cache_ttl": 3600,
        "debug": False,
    }
    manager.save_config_json("environments/production.json", env_config)
    print("Environment configuration saved")

    print("\n=== Questionnaire Config Example ===")
    questionnaire_config = {
        "version": "2.0",
        "questions": [
            {
                "id": "PA01",
                "category": "Policy Alignment",
                "weight": 1.0,
            },
            {
                "id": "PA02",
                "category": "Policy Alignment",
                "weight": 0.8,
            },
        ],
    }
    manager.save_config_json(
        "questionnaire/questionnaire_v2.json", questionnaire_config
    )
    print("Questionnaire configuration saved")

    print("\n=== Final Registry ===")
    final_registry = manager.get_registry()
    print(f"Total files: {len(final_registry)}")
    for path, info in final_registry.items():
        print(f"  {path}")
        print(f"    Hash: {info['hash'][:16]}...")
        print(f"    Modified: {info['last_modified']}")
        print(f"    Size: {info['size_bytes']} bytes")


if __name__ == "__main__":
    main()



================================================================================
FILE: test_factory_refactor.py
================================================================================

#!/usr/bin/env python3
"""Test script to verify factory refactoring."""

import sys
from pathlib import Path

def check_no_direct_imports():
    """Check that API files don’t have direct imports of processing/analysis/utils."""
    errors = []

    # Check pipeline_connector.py
    connector_path = Path('src/farfan_pipeline/api/pipeline_connector.py')
    with open(connector_path, 'r') as f:
        content = f.read()
        if 'from farfan_pipeline.processing.spc_ingestion import' in content:
            errors.append("pipeline_connector.py still has direct spc_ingestion import")
        if 'from farfan_pipeline.utils.spc_adapter import' in content:
            errors.append("pipeline_connector.py still has direct spc_adapter import")
        if 'from farfan_pipeline.utils.cpp_adapter import' in content and 'factory' not in content.split('from farfan_pipeline.utils.cpp_adapter import')[0]
.split('\n')[-1]:
            errors.append("pipeline_connector.py still has direct cpp_adapter import")

    # Check api_server.py
    server_path = Path('src/farfan_pipeline/api/api_server.py')
    with open(server_path, 'r') as f:
        content = f.read()
        if 'from farfan_pipeline.analysis.recommendation_engine import load_recommendation_engine' in content:
            errors.append("api_server.py still has direct recommendation_engine import")

    return errors

def check_factory_methods_used():
    """Check that factory methods are properly used."""
    errors = []

    # Check pipeline_connector.py uses factory methods
    connector_path = Path('src/farfan_pipeline/api/pipeline_connector.py')
    with open(connector_path, 'r') as f:
        content = f.read()
        if 'create_cpp_ingestion_pipeline' not in content:
            errors.append("pipeline_connector.py does not use create_cpp_ingestion_pipeline")
        if 'create_cpp_adapter' not in content:
            errors.append("pipeline_connector.py does not use create_cpp_adapter")
        if 'from ..core.orchestrator.factory import' not in content:
            errors.append("pipeline_connector.py does not import from factory")

    # Check api_server.py uses factory methods
    server_path = Path('src/farfan_pipeline/api/api_server.py')
    with open(server_path, 'r') as f:
        content = f.read()
        if 'create_recommendation_engine' not in content:
            errors.append("api_server.py does not use create_recommendation_engine")
        if 'from farfan_pipeline.core.orchestrator.factory import' not in content:
            errors.append("api_server.py does not import from factory")

    return errors

def check_factory_has_methods():
    """Check that factory.py has the new methods."""
    errors = []

    factory_path = Path('src/farfan_pipeline/core/orchestrator/factory.py')
    with open(factory_path, 'r') as f:
        content = f.read()
        if 'def create_cpp_ingestion_pipeline' not in content:
            errors.append("factory.py missing create_cpp_ingestion_pipeline")
        if 'def create_cpp_adapter' not in content:
            errors.append("factory.py missing create_cpp_adapter")
        if 'def create_recommendation_engine' not in content:
            errors.append("factory.py missing create_recommendation_engine")

    return errors

def main():
    print("Checking factory refactoring...")
    print()

    all_errors = []

    print("1. Checking for direct imports in API layer...")
    errors = check_no_direct_imports()
    if errors:
        all_errors.extend(errors)
        for error in errors:
            print(f"  â\234\227 {error}")
    else:
        print("  â\234\223 No direct imports found")

    print()
    print("2. Checking factory methods are used...")
    errors = check_factory_methods_used()
    if errors:
        all_errors.extend(errors)
        for error in errors:
            print(f"  â\234\227 {error}")
    else:
        print("  â\234\223 Factory methods properly used")

    print()
    print("3. Checking factory has new methods...")
    errors = check_factory_has_methods()
    if errors:
        all_errors.extend(errors)
        for error in errors:
            print(f"  â\234\227 {error}")
    else:
        print("  â\234\223 Factory has all new methods")

    print()
    if all_errors:
        print(f"â\235\214 Refactoring incomplete: {len(all_errors)} errors found")
        return 1
    else:
        print("â\234\205 All refactoring checks passed!")
        return 0

if __name__ == '__main__':
    sys.exit(main())



================================================================================
FILE: test_signal_irrigation_implementation.py
================================================================================

"""
Test Signal Irrigation Implementation
=====================================

Validates that the implemented signal enrichment works correctly
with real data, proper context filtering, and lineage tracking.

NO STUBS. NO PLACEHOLDERS. REAL IMPLEMENTATION ONLY.
"""

import sys
from pathlib import Path

# Add src to path
sys.path.insert(0, str(Path(__file__).parent / "src"))

print("=" * 80)
print("SIGNAL IRRIGATION IMPLEMENTATION TEST")
print("=" * 80)

# Test 1: Import all required modules
print("\n[TEST 1] Importing modules...")
try:
    from farfan_pipeline.core.orchestrator.signal_context_scoper import (
        filter_patterns_by_context,
        create_document_context,
    )
    from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
        create_enriched_signal_pack,
    )
    print("â\234\205 All signal modules imported successfully")
except ImportError as e:
    print(f"â\235\214 Import failed: {e}")
    sys.exit(1)

# Test 2: Verify signal enrichment in flux phase
print("\n[TEST 2] Testing signal enrichment logic...")

# Create mock chunks
mock_chunks = [
    {
        "id": "chunk1",
        "text": "Presupuesto asignado para desarrollo econÃ³mico: COP 1.2M",
        "section": "budget",
        "chapter": 3,
        "policy_area_hint": "PA04",
    },
    {
        "id": "chunk2",
        "text": "Indicador de cobertura territorial especificado",
        "section": "indicators",
        "chapter": 5,
        "policy_area_hint": "PA04",
    },
]

# Create mock signal pack
mock_signal_pack = {
    "patterns": [
        {
            "id": "PAT-001",
            "pattern": "presupuesto",
            "category": "GENERAL",
            "confidence_weight": 0.85,
            "context_requirement": {"section": "budget"},
            "context_scope": "section",
        },
        {
            "id": "PAT-002",
            "pattern": "indicador",
            "category": "GENERAL",
            "confidence_weight": 0.90,
            "context_scope": "global",
        },
        {
            "id": "PAT-003",
            "pattern": "territorial",
            "category": "TERRITORIAL",
            "confidence_weight": 0.80,
            "context_requirement": {"section": "indicators"},
        },
    ],
    "version": "1.0.0",
    "policy_area": "PA04",
}

# Mock registry_get
def mock_registry_get(policy_area):
    if policy_area == "PA04":
        return mock_signal_pack
    return None

# Simulate the enrichment logic from run_signals
enriched_chunks = []
total_patterns = 0

```

```
for chunk in mock_chunks:
    policy_area_hint = chunk.get("policy_area_hint", "default")
    pack = mock_registry_get(policy_area_hint)

    if pack is None:
        enriched_chunks.append({**chunk, "signal_enriched": False})
        continue

    patterns = pack.get("patterns", [])
    doc_context = create_document_context(
        section=chunk.get("section"),
        chapter=chunk.get("chapter"),
        policy_area=policy_area_hint,
    )

    applicable_patterns, stats = filter_patterns_by_context(patterns, doc_context)

    enriched_chunks.append({
        **chunk,
        "signal_enriched": True,
        "applicable_patterns": [
            {
                "pattern_id": p.get("id"),
                "pattern": p.get("pattern"),
                "confidence_weight": p.get("confidence_weight"),
            }
            for p in applicable_patterns
        ],
        "pattern_count": len(applicable_patterns),
        "filtering_stats": stats,
    })

    total_patterns += len(applicable_patterns)

print(f"  Chunks enriched: {len(enriched_chunks)}")
print(f"  Total applicable patterns: {total_patterns}")

# Validate chunk 1 (budget context)
chunk1 = enriched_chunks[0]
assert chunk1["signal_enriched"] == True, "Chunk 1 should be enriched"
assert chunk1["pattern_count"] >= 1, "Chunk 1 should have at least 1 applicable pattern"

# Pattern PAT-001 should match (budget section)
pattern_ids = [p["pattern_id"] for p in chunk1["applicable_patterns"]]
assert "PAT-001" in pattern_ids or "PAT-002" in pattern_ids, "Chunk 1 should have budget or global pattern"

print(f"  Chunk 1 patterns: {chunk1['pattern_count']} applicable")
print(f"    - {chunk1['filtering_stats']['context_filtered']} context-filtered")
print(f"    - {chunk1['filtering_stats']['scope_filtered']} scope-filtered")

# Validate chunk 2 (indicators context)
chunk2 = enriched_chunks[1]
assert chunk2["signal_enriched"] == True, "Chunk 2 should be enriched"
assert chunk2["pattern_count"] >= 1, "Chunk 2 should have at least 1 applicable pattern"

print(f"  Chunk 2 patterns: {chunk2['pattern_count']} applicable")
print(f"    - {chunk2['filtering_stats']['context_filtered']} context-filtered")

print("â\234\205 Signal enrichment working correctly")

# Test 3: Verify lineage tracking in evidence extraction
print("\n[TEST 3] Testing signal lineage tracking...")

from farfan_pipeline.core.orchestrator.signal_evidence_extractor import (
    extract_evidence_for_element_type,
)

text = "Presupuesto municipal asignado: COP 500M para desarrollo econÃ³mico"
patterns = [
    {
        "id": "PAT-BUDGET-001",
        "pattern": "presupuesto",
        "category": "GENERAL",
        "confidence_weight": 0.85,
        "match_type": "substring",
    }
]

matches = extract_evidence_for_element_type(
    element_type="presupuesto_municipal",
    text=text,
    all_patterns=patterns,
    validations={},
)

print(f"  Matches found: {len(matches)}")

if matches:
    match = matches[0]
    print(f"  Match value: '{match['value']}'")
    print(f"  Pattern ID: {match['pattern_id']}")
    print(f"  Confidence: {match['confidence']}")

    # Verify lineage exists
    assert "lineage" in match, "Match should have lineage tracking"
    lineage = match["lineage"]

    print(f"  Lineage:")
    print(f"    - pattern_id: {lineage['pattern_id']}")
    print(f"    - element_type: {lineage['element_type']}")
    print(f"    - extraction_phase: {lineage['extraction_phase']}")
    print(f"    - confidence_weight: {lineage['confidence_weight']}")

    assert lineage["pattern_id"] == "PAT-BUDGET-001", "Lineage pattern_id should match"
    assert lineage["element_type"] == "presupuesto_municipal", "Lineage element_type should match"
    assert lineage["extraction_phase"] == "microanswering", "Lineage phase should be microanswering"

    print("â\234\205 Signal lineage tracking working correctly")
else:
    print("â\232 ï¸\217  No matches found (pattern may not match text)")

# Test 4: Integration test with EnrichedSignalPack
```

```
print("\n[TEST 4] Testing EnrichedSignalPack integration...")

class MockBasePack:
    def __init__(self, patterns):
        self.patterns = patterns
        self.policy_area = "PA04"
        self.version = "1.0.0"

base_pack = MockBasePack(mock_signal_pack["patterns"])
enriched_pack = create_enriched_signal_pack(base_pack, enable_semantic_expansion=False)

# Test context filtering via enriched pack
doc_context = create_document_context(section="budget", chapter=3)
filtered_patterns = enriched_pack.get_patterns_for_context(doc_context)

print(f"  Enriched pack created: {type(enriched_pack).__name__}")
print(f"  Original patterns: {len(base_pack.patterns)}")
print(f"  Context-filtered patterns: {len(filtered_patterns)}")

assert len(filtered_patterns) >= 1, "Should have at least 1 pattern for budget context"
print("â\234\205 EnrichedSignalPack integration working correctly")

# Test 5: Verify NO inappropriate signal bleeding
print("\n[TEST 5] Testing stage-appropriate signal scoping...")

# Scoring signals should NOT appear in chunks
chunk_with_patterns = enriched_chunks[0]
for pattern in chunk_with_patterns["applicable_patterns"]:
    # Verify only extraction-relevant fields, NO scoring fields
    assert "pattern" in pattern, "Pattern should have pattern text (needed for extraction)"
    assert "confidence_weight" in pattern, "Pattern should have confidence (needed for weighting)"
    # These would be inappropriate for chunks:
    assert "scoring_modality" not in pattern, "Chunks should NOT receive scoring_modality"
    assert "failure_contract" not in pattern, "Chunks should NOT receive failure_contract"

print("  â\234\223 No scoring signals in chunks")
print("  â\234\223 No validation signals in chunks")
print("  â\234\223 Only extraction-relevant signals present")
print("â\234\205 Stage-appropriate signal scoping validated")

# Final summary
print("\n" + "=" * 80)
print("ALL TESTS PASSED â\234\205")
print("=" * 80)
print("\nImplementation Summary:")
print("  â\234\223 Real signal enrichment (no stubs)")
print("  â\234\223 Context-aware pattern filtering")
print("  â\234\223 Signal lineage tracking")
print("  â\234\223 EnrichedSignalPack integration")
print("  â\234\223 Stage-appropriate signal scoping")
print("\nIUI Improvement Estimate:")
print("  â\200¢ Chunking: 0% â\206\222 56% (+56 pp)")
print("  â\200¢ MicroAnswering: 30% â\206\222 93% (+63 pp)")
print("  â\200¢ Overall: 26% â\206\222 70% (+44 pp)")
print("\n" + "=" * 80)

```

```


================================================================================
FILE: tests/analysis/__init__.py
================================================================================

"""Analysis layer test suite."""



================================================================================
FILE: tests/analysis/test_bayesian_multilevel.py
================================================================================

"""
Unit tests for Bayesian Multilevel System (bayesian_multilevel_system.py)

Tests reconciliation validators, Bayesian updaters, probative tests,
dispersion engine, and posterior sampling validation.
"""
from unittest.mock import Mock

import pytest

from src.farfan_pipeline.analysis.bayesian_multilevel_system import (
    BayesianUpdater,
    DispersionEngine,
    MicroLevelAnalysis,
    ProbativeTest,
    ProbativeTestType,
    ReconciliationValidator,
    ValidationResult,
    ValidationRule,
    ValidatorType,
)


@pytest.fixture
def sample_validation_rules():
    """Create sample validation rules."""
    return [
        ValidationRule(
            validator_type=ValidatorType.RANGE,
            field_name='score',
            expected_range=(0.0, 100.0),
            penalty_factor=0.1
        ),
        ValidationRule(
            validator_type=ValidatorType.UNIT,
            field_name='unit',
            expected_unit='porcentaje',
            penalty_factor=0.05
        ),
        ValidationRule(
            validator_type=ValidatorType.PERIOD,
            field_name='period',
            expected_period='2024',
            penalty_factor=0.05
        )
    ]


@pytest.fixture
def reconciliation_validator(sample_validation_rules):
    """Create reconciliation validator."""
    return ReconciliationValidator(sample_validation_rules)


@pytest.fixture
def bayesian_updater():
    """Create Bayesian updater."""
    return BayesianUpdater()


class TestReconciliationValidator:
    """Test suite for ReconciliationValidator."""

    def test_validate_range_within_bounds(self, reconciliation_validator):
        """Test range validation with value within bounds."""
        rule = ValidationRule(
            validator_type=ValidatorType.RANGE,
            field_name='score',
            expected_range=(0.0, 100.0),
            penalty_factor=0.1
        )

        result = reconciliation_validator.validate_range(50.0, rule)

        assert result.passed is True
        assert result.violation_severity == 0.0
        assert result.penalty_applied == 0.0

    def test_validate_range_outside_bounds(self, reconciliation_validator):
        """Test range validation with value outside bounds."""
        rule = ValidationRule(
            validator_type=ValidatorType.RANGE,
            field_name='score',
            expected_range=(0.0, 100.0),
            penalty_factor=0.1
        )

        result = reconciliation_validator.validate_range(150.0, rule)

        assert result.passed is False
        assert result.violation_severity > 0.0
        assert result.penalty_applied > 0.0

    def test_validate_unit_matching(self, reconciliation_validator):
        """Test unit validation with matching unit."""
        rule = ValidationRule(
            validator_type=ValidatorType.UNIT,
            field_name='unit',
            expected_unit='porcentaje',
            penalty_factor=0.05
        )

        result = reconciliation_validator.validate_unit('porcentaje', rule)

        assert result.passed is True
        assert result.penalty_applied == 0.0

    def test_validate_unit_not_matching(self, reconciliation_validator):
        """Test unit validation with non-matching unit."""
        rule = ValidationRule(
            validator_type=ValidatorType.UNIT,
            field_name='unit',
            expected_unit='porcentaje',
            penalty_factor=0.05
        )

        result = reconciliation_validator.validate_unit('numero', rule)

        assert result.passed is False
        assert result.violation_severity == 1.0
        assert result.penalty_applied > 0.0

    def test_validate_data_multiple_rules(self, reconciliation_validator):
        """Test validation of data against multiple rules."""
        data = {
            'score': 85.0,
            'unit': 'porcentaje',
            'period': '2024'
        }

        results = reconciliation_validator.validate_data(data)

        assert isinstance(results, list)
        assert len(results) == 3
        assert all(r.passed for r in results)

    def test_calculate_total_penalty(self, reconciliation_validator):
        """Test total penalty calculation."""
        results = [
            ValidationResult(
                rule=Mock(), passed=False, observed_value=150,
                expected_value=(0, 100), violation_severity=0.5,
                penalty_applied=0.05
            ),
            ValidationResult(
                rule=Mock(), passed=False, observed_value='wrong',
                expected_value='correct', violation_severity=1.0,
                penalty_applied=0.1
            )
        ]

        total_penalty = reconciliation_validator.calculate_total_penalty(results)

        assert total_penalty == 0.15
```

```


class TestBayesianUpdater:
    """Test suite for BayesianUpdater."""

    def test_initialization(self, bayesian_updater):
        """Test Bayesian updater initialization."""
        assert bayesian_updater is not None
        assert len(bayesian_updater.updates) == 0

    def test_update_with_straw_in_wind(self, bayesian_updater):
        """Test Bayesian update with straw-in-wind test."""
        test = ProbativeTest(
            test_type=ProbativeTestType.STRAW_IN_WIND,
            test_name='Weak evidence test',
            evidence_strength=0.6,
            prior_probability=0.5
        )

        posterior = bayesian_updater.update(0.5, test, test_passed=True)

        assert 0.0 <= posterior <= 1.0
        assert len(bayesian_updater.updates) == 1

    def test_update_with_hoop_test_pass(self, bayesian_updater):
        """Test Bayesian update with passing hoop test."""
        test = ProbativeTest(
            test_type=ProbativeTestType.HOOP_TEST,
            test_name='Necessary condition test',
            evidence_strength=0.8,
            prior_probability=0.5
        )

        posterior = bayesian_updater.update(0.5, test, test_passed=True)

        assert posterior > 0.0
        assert posterior <= 1.0

    def test_sequential_update(self, bayesian_updater):
        """Test sequential Bayesian updating."""
        tests = [
            (ProbativeTest(
                test_type=ProbativeTestType.STRAW_IN_WIND,
                test_name='Test 1',
                evidence_strength=0.6,
                prior_probability=0.5
            ), True),
            (ProbativeTest(
                test_type=ProbativeTestType.HOOP_TEST,
                test_name='Test 2',
                evidence_strength=0.8,
                prior_probability=0.5
            ), True)
        ]

        final_posterior = bayesian_updater.sequential_update(0.5, tests)
        assert 0.0 <= final_posterior <= 1.0
        assert len(bayesian_updater.updates) == 2

    def test_export_to_csv(self, bayesian_updater, tmp_path):
        """Test exporting posterior table to CSV."""
        test = ProbativeTest(
            test_type=ProbativeTestType.STRAW_IN_WIND,
            test_name='Test',
            evidence_strength=0.6,
            prior_probability=0.5
        )
        bayesian_updater.update(0.5, test, test_passed=True)

        output_path = tmp_path / 'posterior_table_micro.csv'
        bayesian_updater.export_to_csv(output_path)

        assert output_path.exists()


class TestDispersionEngine:
    """Test suite for DispersionEngine."""

    @pytest.fixture
    def dispersion_engine(self):
        """Create dispersion engine."""
        return DispersionEngine(dispersion_threshold=0.3)

    def test_calculate_cv(self, dispersion_engine):
        """Test coefficient of variation calculation."""
        scores = [80.0, 85.0, 90.0, 95.0, 100.0]
        cv = dispersion_engine.calculate_cv(scores)

        assert isinstance(cv, float)
        assert cv >= 0.0

    def test_calculate_cv_empty_list(self, dispersion_engine):
        """Test CV with empty list."""
        cv = dispersion_engine.calculate_cv([])
        assert cv == 0.0

    def test_calculate_max_gap(self, dispersion_engine):
        """Test maximum gap calculation."""
        scores = [60.0, 70.0, 85.0, 95.0]
        max_gap = dispersion_engine.calculate_max_gap(scores)

        assert isinstance(max_gap, float)
        assert max_gap > 0.0

    def test_calculate_gini(self, dispersion_engine):
        """Test Gini coefficient calculation."""
        scores = [50.0, 60.0, 70.0, 80.0, 90.0]
        gini = dispersion_engine.calculate_gini(scores)

        assert isinstance(gini, float)
        assert 0.0 <= gini <= 1.0
```

```


class TestIntegration:
    """Integration tests for Bayesian multilevel system."""

    def test_micro_level_analysis_pipeline(self, reconciliation_validator, bayesian_updater):
        """Test complete MICRO level analysis pipeline."""
        # Validation
        data = {'score': 75.0, 'unit': 'porcentaje', 'period': '2024'}
        validation_results = reconciliation_validator.validate_data(data)
        validation_penalty = reconciliation_validator.calculate_total_penalty(validation_results)

        # Bayesian updating
        test = ProbativeTest(
            test_type=ProbativeTestType.HOOP_TEST,
            test_name='Quality test',
            evidence_strength=0.8,
            prior_probability=0.5
        )
        final_posterior = bayesian_updater.update(0.5, test, test_passed=True)

        # Create MicroLevelAnalysis
        analysis = MicroLevelAnalysis(
            question_id='Q001',
            raw_score=75.0,
            validation_results=validation_results,
            validation_penalty=validation_penalty,
            bayesian_updates=bayesian_updater.updates,
            final_posterior=final_posterior,
            adjusted_score=75.0 - validation_penalty
        )

        assert analysis.question_id == 'Q001'
        assert analysis.final_posterior == final_posterior
        assert analysis.adjusted_score <= analysis.raw_score



================================================================================
FILE: tests/analysis/test_contradiction_detector.py
================================================================================

"""
Unit tests for PolicyContradictionDetector (contradiction_deteccion.py)

Tests semantic contradiction detection, temporal logic verification,
Bayesian confidence calculation, and contradiction evidence generation.
"""
from unittest.mock import Mock, patch

import numpy as np
import pytest

from src.farfan_pipeline.analysis.contradiction_deteccion import (
    BayesianConfidenceCalculator,
    ContradictionEvidence,
    ContradictionType,
    PolicyContradictionDetector,
    PolicyDimension,
    PolicyStatement,
    TemporalLogicVerifier,
)


@pytest.fixture
def bayesian_calculator():
    """Create Bayesian confidence calculator."""
    return BayesianConfidenceCalculator()


@pytest.fixture
def temporal_verifier():
    """Create temporal logic verifier."""
    return TemporalLogicVerifier()


@pytest.fixture
def sample_policy_statements():
    """Create sample policy statements for testing."""
    return [
        PolicyStatement(
            text="El presupuesto para educaciÃ³n es de $500 millones en 2024",
            dimension=PolicyDimension.FINANCIERO,
            position=(0, 100),
            temporal_markers=["2024"],
            quantitative_claims=[{"amount": 500, "unit": "millones"}]
        ),
        PolicyStatement(
            text="El presupuesto educativo alcanzarÃ¡ $600 millones en 2024",
            dimension=PolicyDimension.FINANCIERO,
            position=(200, 300),
            temporal_markers=["2024"],
            quantitative_claims=[{"amount": 600, "unit": "millones"}]
        ),
        PolicyStatement(
            text="La meta es alcanzar 100% cobertura antes de 2023",
            dimension=PolicyDimension.ESTRATEGICO,
            position=(400, 500),
            temporal_markers=["2023"],
            quantitative_claims=[{"percentage": 100}]
        )
    ]


class TestBayesianConfidenceCalculator:
    """Test suite for Bayesian confidence calculator."""

    def test_initialization(self, bayesian_calculator):
        """Test calculator initialization with domain priors."""
        assert bayesian_calculator.prior_alpha == 2.5
        assert bayesian_calculator.prior_beta == 7.5

    def test_calculate_posterior_basic(self, bayesian_calculator):
        """Test basic posterior probability calculation."""
        posterior = bayesian_calculator.calculate_posterior(
            evidence_strength=0.8,
            observations=5,
            domain_weight=1.0
        )

        assert isinstance(posterior, float)
        assert 0.0 <= posterior <= 1.0
        assert posterior > 0.0

    def test_calculate_posterior_weak_evidence(self, bayesian_calculator):
        """Test posterior with weak evidence."""
        posterior = bayesian_calculator.calculate_posterior(
            evidence_strength=0.2,
            observations=2,
            domain_weight=1.0
        )

        assert isinstance(posterior, float)
        assert 0.0 <= posterior <= 1.0

    def test_calculate_posterior_strong_evidence(self, bayesian_calculator):
        """Test posterior with strong evidence."""
        posterior = bayesian_calculator.calculate_posterior(
            evidence_strength=0.95,
            observations=10,
            domain_weight=1.0
        )

        assert isinstance(posterior, float)
        assert posterior > 0.5

    def test_calculate_posterior_domain_weight(self, bayesian_calculator):
        """Test domain weight effect on posterior."""
        posterior_high_weight = bayesian_calculator.calculate_posterior(
            evidence_strength=0.7,
            observations=5,
            domain_weight=2.0
        )

        posterior_low_weight = bayesian_calculator.calculate_posterior(
            evidence_strength=0.7,
            observations=5,
            domain_weight=0.5
        )

        # Higher domain weight should increase confidence
        assert posterior_high_weight >= posterior_low_weight

    def test_posterior_bounds(self, bayesian_calculator):
        """Test posterior is always bounded [0, 1]."""
        for evidence in [0.0, 0.5, 1.0]:
            for obs in [1, 5, 10]:
                posterior = bayesian_calculator.calculate_posterior(
                    evidence_strength=evidence,
                    observations=obs
                )
                assert 0.0 <= posterior <= 1.0


class TestTemporalLogicVerifier:
    """Test suite for temporal logic verification."""

    def test_initialization(self, temporal_verifier):
        """Test temporal verifier initialization."""
        assert hasattr(temporal_verifier, ’temporal_patterns’)
        assert ’sequential’ in temporal_verifier.temporal_patterns
        assert ’parallel’ in temporal_verifier.temporal_patterns
        assert ’deadline’ in temporal_verifier.temporal_patterns

    def test_verify_temporal_consistency_no_conflicts(self, temporal_verifier):
        """Test temporal verification with consistent statements."""
        statements = [
            PolicyStatement(
                text="Primero el diagnÃ³stico",
                dimension=PolicyDimension.DIAGNOSTICO,
                position=(0, 50),
                temporal_markers=["primero"]
            ),
            PolicyStatement(
                text="Luego la implementaciÃ³n",
                dimension=PolicyDimension.PROGRAMATICO,
                position=(100, 150),
                temporal_markers=["luego"]
            )
        ]

        is_consistent, conflicts = temporal_verifier.verify_temporal_consistency(statements)

        assert isinstance(is_consistent, bool)
        assert isinstance(conflicts, list)

    def test_parse_temporal_marker_year(self, temporal_verifier):
        """Test parsing year from temporal marker."""
        marker = "aÃ±o 2024"
        timestamp = temporal_verifier._parse_temporal_marker(marker)

        assert timestamp == 2024

    def test_parse_temporal_marker_quarter(self, temporal_verifier):
        """Test parsing quarter from temporal marker."""
        marker = "Q2"
        timestamp = temporal_verifier._parse_temporal_marker(marker)

        assert timestamp == 2

    def test_parse_temporal_marker_spanish_quarter(self, temporal_verifier):
        """Test parsing Spanish quarter markers."""
        marker = "segundo trimestre"
        timestamp = temporal_verifier._parse_temporal_marker(marker)

        assert timestamp == 2

    def test_classify_temporal_type(self, temporal_verifier):
        """Test temporal marker type classification."""
        sequential_marker = "primero establecer"
        parallel_marker = "simultÃ¡neamente ejecutar"
        deadline_marker = "antes de finalizar"

        assert temporal_verifier._classify_temporal_type(sequential_marker) == 'sequential'
        assert temporal_verifier._classify_temporal_type(parallel_marker) == 'parallel'
        assert temporal_verifier._classify_temporal_type(deadline_marker) == 'deadline'

    def test_extract_resources(self, temporal_verifier):
        """Test resource extraction from text."""
        text = "asignar presupuesto y recursos humanos para el proyecto"
        resources = temporal_verifier._extract_resources(text)

        assert isinstance(resources, list)
        assert len(resources) > 0


class TestPolicyContradictionDetector:
    """Test suite for policy contradiction detector."""

    @pytest.fixture
    def detector(self):
        """Create contradiction detector with mocked models."""
        with patch('src.farfan_pipeline.analysis.contradiction_deteccion.SentenceTransformer'), \
             patch('src.farfan_pipeline.analysis.contradiction_deteccion.pipeline'), \
             patch('src.farfan_pipeline.analysis.factory.load_spacy_model'):

            detector = PolicyContradictionDetector()

            # Mock the semantic model
            detector.semantic_model = Mock()
            detector.semantic_model.encode = Mock(return_value=np.random.rand(768))

            # Mock NLP model
            detector.nlp = Mock()

            return detector

    def test_initialization(self, detector):
        """Test detector initialization."""
        assert hasattr(detector, 'semantic_model')
        assert hasattr(detector, 'bayesian_calculator')
        assert hasattr(detector, 'temporal_verifier')
        assert hasattr(detector, 'knowledge_graph')

    def test_detect_with_empty_text(self, detector):
        """Test detection with empty text."""
        result = detector.detect("", plan_name="Test Plan")

        assert isinstance(result, dict)
        assert "contradictions" in result
        assert "total_contradictions" in result

    def test_detect_with_simple_text(self, detector):
        """Test detection with simple non-contradictory text."""
        text = "El plan de desarrollo busca mejorar la educaciÃ³n y la salud."
        result = detector.detect(text, plan_name="Test Plan")

        assert isinstance(result, dict)
        assert "contradictions" in result
        assert "coherence_metrics" in result
        assert result["plan_name"] == "Test Plan"

    def test_extract_policy_statements(self, detector):
        """Test policy statement extraction."""
        text = """
        El presupuesto es de $500 millones.
        La meta es alcanzar 100% cobertura.
        """

        with patch.object(detector, '_extract_policy_statements') as mock_extract:
            mock_extract.return_value = [
                PolicyStatement(
                    text="El presupuesto es de $500 millones",
                    dimension=PolicyDimension.FINANCIERO,
                    position=(0, 50)
                )
            ]

            statements = detector._extract_policy_statements(text, PolicyDimension.FINANCIERO)
            assert len(statements) > 0
            assert isinstance(statements[0], PolicyStatement)

    def test_contradiction_type_classification(self, detector):
        """Test contradiction type is properly classified."""
        # Ensure ContradictionType enum is used correctly
        assert ContradictionType.NUMERICAL_INCONSISTENCY
        assert ContradictionType.TEMPORAL_CONFLICT
        assert ContradictionType.SEMANTIC_OPPOSITION
        assert ContradictionType.LOGICAL_INCOMPATIBILITY

    def test_serialize_contradiction(self, detector):
        """Test contradiction evidence serialization."""
        statement_a = PolicyStatement(
            text="Statement A",
            dimension=PolicyDimension.ESTRATEGICO,
            position=(0, 10)
        )
        statement_b = PolicyStatement(
            text="Statement B",
            dimension=PolicyDimension.ESTRATEGICO,
            position=(20, 30)
        )

        evidence = ContradictionEvidence(
            statement_a=statement_a,
            statement_b=statement_b,
            contradiction_type=ContradictionType.SEMANTIC_OPPOSITION,
            confidence=0.85,
            severity=0.7,
            semantic_similarity=0.3,
            logical_conflict_score=0.8,
            temporal_consistency=True,
            numerical_divergence=None,
            affected_dimensions=[PolicyDimension.ESTRATEGICO],
            resolution_suggestions=["Review strategic alignment"]
        )

        serialized = detector._serialize_contradiction(evidence)

        assert isinstance(serialized, dict)
        assert "contradiction_type" in serialized
        assert "confidence" in serialized
        assert "severity" in serialized


class TestContradictionDetectionIntegration:
    """Integration tests for contradiction detection."""

    @pytest.fixture
    def detector_integration(self):
        """Create detector for integration tests."""
        with patch('src.farfan_pipeline.analysis.contradiction_deteccion.SentenceTransformer'), \
             patch('src.farfan_pipeline.analysis.contradiction_deteccion.pipeline'), \
             patch('src.farfan_pipeline.analysis.factory.load_spacy_model'):

            detector = PolicyContradictionDetector()
            detector.semantic_model = Mock()
            detector.semantic_model.encode = Mock(return_value=np.random.rand(768))
            detector.nlp = Mock()

            return detector

    def test_numerical_contradiction_detection(self, detector_integration):
        """Test detection of numerical contradictions."""
        text = """
        El presupuesto para educaciÃ³n es de 500 millones de pesos en 2024.
        El mismo presupuesto educativo alcanzarÃ¡ 600 millones en 2024.
        """

        result = detector_integration.detect(text, plan_name="Test")

        assert isinstance(result, dict)
        assert "contradictions" in result

    def test_temporal_contradiction_detection(self, detector_integration):
        """Test detection of temporal contradictions."""
        text = """
        La meta debe alcanzarse antes de 2023.
        El programa comenzarÃ¡ despuÃ©s de 2024.
        """

        result = detector_integration.detect(text, plan_name="Test")

        assert isinstance(result, dict)
        assert "contradictions" in result

    def test_coherence_metrics_calculation(self, detector_integration):
        """Test coherence metrics are calculated."""
        text = "Plan de desarrollo con mÃºltiples componentes estratÃ©gicos."

        result = detector_integration.detect(text, plan_name="Test")

        assert "coherence_metrics" in result
        assert isinstance(result["coherence_metrics"], dict)

    def test_recommendations_generation(self, detector_integration):
        """Test resolution recommendations are generated."""
        text = """
        El plan tiene objetivos contradictorios.
        Las metas no son alcanzables simultÃ¡neamente.
        """

        result = detector_integration.detect(text, plan_name="Test")

        assert "recommendations" in result or "contradictions" in result



================================================================================
FILE: tests/analysis/test_derek_beach_causal.py
================================================================================

"""
Unit tests for Derek Beach Causal Extractor (derek_beach.py)

Tests mechanistic evidence evaluation, Beach evidential tests taxonomy,
entity-activity extraction, and causal DAG construction.
"""

import networkx as nx
import pytest

from src.farfan_pipeline.analysis.derek_beach import (
    BeachEvidentialTest,
    CausalLink,
    CDAFException,
    ConfigLoader,
    EntityActivity,
    MetaNode,
)


@pytest.fixture
def mock_config():
    """Create mock configuration for testing."""
    config = {
        'patterns': {
            'goal_codes': r'[MP][RIP]-\d{3}',
            'numeric_formats': r'[\d,]+(?:\.\d+)?%?'
        },
        'lexicons': {
            'causal_logic': ['mediante', 'a travÃ©s de', 'para lograr'],
            'contextual_factors': ['riesgo', 'amenaza', 'limitaciÃ³n']
        },
        'entity_aliases': {
            'SEC GOB': 'SecretarÃa de Gobierno'
        },
        'bayesian_thresholds': {
            'kl_divergence': 0.01,
            'prior_alpha': 2.0,
            'prior_beta': 2.0
        }
    }
    return config


@pytest.fixture
def sample_meta_node():
    """Create sample MetaNode for testing."""
    return MetaNode(
        id='MR-001',
        text='Incrementar cobertura educativa al 95%',
        type='resultado',
        baseline=85.0,
        target=95.0,
        unit='porcentaje',
        responsible_entity='SecretarÃa de EducaciÃ³n',
        rigor_status='fuerte',
        dynamics='suma'
    )


class TestBeachEvidentialTest:
    """Test suite for Beach evidential test taxonomy."""

    def test_classify_hoop_test(self):
        """Test classification of hoop test (high necessity, low sufficiency)."""
        test_type = BeachEvidentialTest.classify_test(necessity=0.8, sufficiency=0.3)
        assert test_type == 'hoop_test'

    def test_classify_smoking_gun(self):
        """Test classification of smoking gun test (low necessity, high sufficiency)."""
        test_type = BeachEvidentialTest.classify_test(necessity=0.3, sufficiency=0.8)
        assert test_type == 'smoking_gun'

    def test_classify_doubly_decisive(self):
        """Test classification of doubly decisive test (both high)."""
        test_type = BeachEvidentialTest.classify_test(necessity=0.9, sufficiency=0.9)
        assert test_type == 'doubly_decisive'

    def test_classify_straw_in_wind(self):
        """Test classification of straw-in-wind test (both low)."""
        test_type = BeachEvidentialTest.classify_test(necessity=0.3, sufficiency=0.3)
        assert test_type == 'straw_in_wind'

    def test_apply_hoop_test_failure(self):
        """Test hoop test failure eliminates hypothesis."""
        posterior, interpretation = BeachEvidentialTest.apply_test_logic(
            test_type='hoop_test',
            evidence_found=False,
            prior=0.5,
            bayes_factor=2.0
        )

        assert posterior <= 0.05
        assert 'HOOP_TEST_FAILURE' in interpretation
        assert 'eliminated' in interpretation.lower()

    def test_apply_hoop_test_pass(self):
        """Test hoop test pass allows hypothesis to survive."""
        posterior, interpretation = BeachEvidentialTest.apply_test_logic(
            test_type='hoop_test',
            evidence_found=True,
            prior=0.5,
            bayes_factor=2.0
        )

        assert posterior > 0.0
        assert 'HOOP_TEST_PASSED' in interpretation

    def test_apply_smoking_gun_found(self):
        """Test smoking gun found strongly confirms hypothesis."""
        posterior, interpretation = BeachEvidentialTest.apply_test_logic(
            test_type='smoking_gun',
            evidence_found=True,
            prior=0.3,
            bayes_factor=5.0
        )

        assert posterior > 0.5
        assert 'SMOKING_GUN_FOUND' in interpretation

    def test_apply_doubly_decisive_confirmed(self):
        """Test doubly decisive confirmation is conclusive."""
        posterior, interpretation = BeachEvidentialTest.apply_test_logic(
            test_type='doubly_decisive',
            evidence_found=True,
            prior=0.5,
            bayes_factor=10.0
        )

        assert posterior >= 0.95
        assert 'DOUBLY_DECISIVE_CONFIRMED' in interpretation

    def test_apply_doubly_decisive_eliminated(self):
        """Test doubly decisive elimination is conclusive."""
        posterior, interpretation = BeachEvidentialTest.apply_test_logic(
            test_type='doubly_decisive',
            evidence_found=False,
            prior=0.5,
            bayes_factor=10.0
        )

        assert posterior <= 0.05
        assert 'DOUBLY_DECISIVE_ELIMINATED' in interpretation


class TestMetaNode:
    """Test suite for MetaNode data structure."""

    def test_meta_node_creation(self, sample_meta_node):
        """Test MetaNode instantiation."""
        assert sample_meta_node.id == 'MR-001'
        assert sample_meta_node.type == 'resultado'
        assert sample_meta_node.baseline == 85.0
        assert sample_meta_node.target == 95.0
        assert sample_meta_node.rigor_status == 'fuerte'

    def test_meta_node_entity_activity(self):
        """Test EntityActivity association with MetaNode."""
        entity_activity = EntityActivity(
            entity='SecretarÃ-a de EducaciÃ³n',
            activity='implementar programa',
            verb_lemma='implementar',
            confidence=0.85
        )

        node = MetaNode(
            id='MP-001',
            text='Implementar programa educativo',
            type='producto',
            entity_activity=entity_activity
        )

        assert node.entity_activity is not None
        assert node.entity_activity.entity == 'SecretarÃ-a de EducaciÃ³n'
        assert node.entity_activity.confidence == 0.85

    def test_meta_node_audit_flags(self, sample_meta_node):
        """Test audit flags mechanism."""
        sample_meta_node.audit_flags.append('missing_baseline')
        sample_meta_node.audit_flags.append('unclear_responsibility')

        assert len(sample_meta_node.audit_flags) == 2
        assert 'missing_baseline' in sample_meta_node.audit_flags

    def test_meta_node_confidence_score(self, sample_meta_node):
        """Test confidence score is bounded [0, 1]."""
        sample_meta_node.confidence_score = 0.95
        assert 0.0 <= sample_meta_node.confidence_score <= 1.0


class TestEntityActivity:
    """Test suite for EntityActivity tuple."""

    def test_entity_activity_creation(self):
        """Test EntityActivity instantiation."""
        ea = EntityActivity(
            entity='SecretarÃ-a de Salud',
            activity='gestionar recursos',
            verb_lemma='gestionar',
            confidence=0.9
        )

        assert ea.entity == 'SecretarÃ-a de Salud'
        assert ea.activity == 'gestionar recursos'
        assert ea.verb_lemma == 'gestionar'
        assert ea.confidence == 0.9

    def test_entity_activity_confidence_bounds(self):
        """Test confidence is properly bounded."""
        ea = EntityActivity(
            entity='Test Entity',
            activity='test activity',
            verb_lemma='test',
            confidence=0.75
        )

        assert 0.0 <= ea.confidence <= 1.0


class TestCausalLink:
    """Test suite for CausalLink structure."""

    def test_causal_link_creation(self):
        """Test CausalLink dictionary structure."""
        link: CausalLink = {
            'source': 'MP-001',
            'target': 'MR-001',
            'logic': 'mediante',
            'strength': 0.8,
            'evidence': ['evidencia textual 1', 'evidencia textual 2'],
            'posterior_mean': 0.75,
            'posterior_std': 0.1,
            'kl_divergence': 0.02,
            'converged': True
        }

        assert link['source'] == 'MP-001'
        assert link['target'] == 'MR-001'
        assert link['logic'] == 'mediante'
        assert 0.0 <= link['strength'] <= 1.0
        assert link['converged'] is True

    def test_causal_link_bayesian_fields(self):
        """Test Bayesian inference fields in CausalLink."""
        link: CausalLink = {
            'source': 'A',
            'target': 'B',
            'logic': 'causa',
            'strength': 0.9,
            'evidence': [],
            'posterior_mean': 0.85,
            'posterior_std': 0.05,
            'kl_divergence': 0.01,
            'converged': True
        }

        assert link['posterior_mean'] is not None
        assert link['posterior_std'] is not None
        assert link['kl_divergence'] < 0.05


class TestConfigLoader:
    """Test suite for configuration loading."""

    @pytest.fixture
    def temp_config_file(self, tmp_path, mock_config):
        """Create temporary config file for testing."""
        import yaml

        config_path = tmp_path / 'test_config.yaml'
        with open(config_path, 'w') as f:
            yaml.dump(mock_config, f)

        return config_path

    def test_config_loader_initialization(self, temp_config_file):
        """Test ConfigLoader loads YAML configuration."""
        loader = ConfigLoader(temp_config_file)

        assert loader.config is not None
        assert 'patterns' in loader.config
        assert 'lexicons' in loader.config

    def test_config_loader_validation(self, temp_config_file):
        """Test configuration validation with Pydantic."""
        loader = ConfigLoader(temp_config_file)

        assert loader.validated_config is not None
        assert hasattr(loader.validated_config, 'patterns')
        assert hasattr(loader.validated_config, 'lexicons')

    def test_config_loader_default_fallback(self, tmp_path):
        """Test default configuration fallback."""
        non_existent_path = tmp_path / 'nonexistent.yaml'
        loader = ConfigLoader(non_existent_path)

        # Should load default config without crashing
        assert loader.config is not None


class TestCDAFException:
    """Test suite for CDAF exception handling."""

    def test_cdaf_exception_creation(self):
        """Test CDAF exception with structured details."""
        exception = CDAFException(
            message='Test error',
            details={'context': 'test context'},
            stage='extraction',
            recoverable=True
        )

        assert exception.message == 'Test error'
        assert exception.details['context'] == 'test context'
        assert exception.stage == 'extraction'
        assert exception.recoverable is True

    def test_cdaf_exception_to_dict(self):
        """Test exception serialization to dictionary."""
        exception = CDAFException(
            message='Serialization test',
            details={'key': 'value'},
            stage='processing'
        )

        exception_dict = exception.to_dict()

        assert isinstance(exception_dict, dict)
        assert exception_dict['message'] == 'Serialization test'
        assert exception_dict['stage'] == 'processing'
        assert 'error_type' in exception_dict


class TestCausalExtractionIntegration:
    """Integration tests for causal mechanism extraction."""

    def test_entity_activity_extraction_from_text(self):
        """Test extraction of entity-activity pairs from policy text."""
        text = "La SecretarÃa de EducaciÃ³n implementarÃ¡ el programa educativo"

        # This would use the actual extractor in integration
        # For unit test, we validate the data structure
        ea = EntityActivity(
            entity='SecretarÃa de EducaciÃ³n',
            activity='implementar programa',
            verb_lemma='implementar',
            confidence=0.8
        )

        assert ea.entity is not None
        assert ea.activity is not None
        assert ea.verb_lemma is not None

    def test_causal_dag_construction(self):
        """Test construction of causal DAG from nodes and links."""
        nodes = {
            'A': MetaNode(id='A', text='Node A', type='producto'),
            'B': MetaNode(id='B', text='Node B', type='resultado')
        }

        links = [
            {
                'source': 'A',
                'target': 'B',
                'logic': 'mediante',
                'strength': 0.8,
                'evidence': [],
                'posterior_mean': None,
                'posterior_std': None,
                'kl_divergence': None,
                'converged': None
            }
        ]

        # Validate DAG structure
        G = nx.DiGraph()
        for node_id in nodes:
            G.add_node(node_id)
        for link in links:
            G.add_edge(link['source'], link['target'])

        assert nx.is_directed_acyclic_graph(G)
        assert G.number_of_nodes() == 2
        assert G.number_of_edges() == 1



================================================================================
FILE: tests/analysis/test_financial_viability.py
================================================================================

"""
Unit tests for Financial Viability Tables (financiero_viabilidad_tablas.py)

Tests PDET municipal plan analysis, table extraction, financial indicator extraction,
and quality scoring with causal inference.
"""
from decimal import Decimal
from unittest.mock import Mock, patch

import numpy as np
import pandas as pd
import pytest

from src.farfan_pipeline.analysis.financiero_viabilidad_tablas import (
    ColombianMunicipalContext,
    ExtractedTable,
    FinancialIndicator,
    PDETMunicipalPlanAnalyzer,
    QualityScore,
    ResponsibleEntity,
)


@pytest.fixture
def mock_analyzer():
    """Create mock analyzer with mocked dependencies."""
    with patch('src.farfan_pipeline.analysis.financiero_viabilidad_tablas.SentenceTransformer'), \
         patch('src.farfan_pipeline.analysis.factory.load_spacy_model'), \
         patch('src.farfan_pipeline.analysis.financiero_viabilidad_tablas.pipeline'):

        analyzer = PDETMunicipalPlanAnalyzer(use_gpu=False)

        # Mock the semantic model
        analyzer.semantic_model = Mock()
        analyzer.semantic_model.encode = Mock(return_value=np.random.rand(768))

        # Mock NLP model
        analyzer.nlp = Mock()

        return analyzer


@pytest.fixture
def sample_dataframe():
    """Create sample DataFrame for testing."""
    return pd.DataFrame({
        'Programa': ['EducaciÃ³n', 'Salud', 'Infraestructura'],
        'Presupuesto': ['500,000,000', '300,000,000', '700,000,000'],
        'Responsable': ['SecretarÃa de EducaciÃ³n', 'SecretarÃa de Salud', 'SecretarÃa de Obras'],
        'Meta': ['95%', '100%', '80%']
    })


class TestColombianMunicipalContext:
    """Test suite for Colombian municipal context."""

    def test_official_systems_defined(self):
        """Test official systems are properly defined."""
        context = ColombianMunicipalContext()

        assert 'SISBEN' in context.OFFICIAL_SYSTEMS
        assert 'SGP' in context.OFFICIAL_SYSTEMS
        assert 'SGR' in context.OFFICIAL_SYSTEMS
        assert 'DANE' in context.OFFICIAL_SYSTEMS

    def test_territorial_categories(self):
        """Test territorial categories are defined."""
        context = ColombianMunicipalContext()

        assert len(context.TERRITORIAL_CATEGORIES) == 7
        assert 1 in context.TERRITORIAL_CATEGORIES
        assert context.TERRITORIAL_CATEGORIES[1]['name'] == 'Especial'

    def test_dnp_dimensions(self):
        """Test DNP dimensions are defined."""
        context = ColombianMunicipalContext()

        assert len(context.DNP_DIMENSIONS) == 5
        assert 'DimensiÃ³n EconÃ³mica' in context.DNP_DIMENSIONS
        assert 'DimensiÃ³n Social' in context.DNP_DIMENSIONS

    def test_pdet_pillars(self):
        """Test PDET pillars are defined."""
        context = ColombianMunicipalContext()

        assert len(context.PDET_PILLARS) == 8
        assert 'Ordenamiento social de la propiedad rural' in context.PDET_PILLARS
        assert 'Salud rural' in context.PDET_PILLARS

    def test_pdet_theory_of_change(self):
        """Test PDET theory of change structure."""
        context = ColombianMunicipalContext()

        assert 'Salud rural' in context.PDET_THEORY_OF_CHANGE
        toc = context.PDET_THEORY_OF_CHANGE['Salud rural']
        assert 'outcomes' in toc
        assert 'mediators' in toc
        assert 'lag_years' in toc


class TestPDETMunicipalPlanAnalyzer:
    """Test suite for PDET Municipal Plan Analyzer."""

    def test_initialization(self, mock_analyzer):
        """Test analyzer initialization."""
        assert mock_analyzer is not None
        assert mock_analyzer.device in ['cuda', 'cpu']
        assert mock_analyzer.context is not None

    def test_clean_dataframe(self, mock_analyzer, sample_dataframe):
        """Test DataFrame cleaning."""
        cleaned = mock_analyzer._clean_dataframe(sample_dataframe)

        assert isinstance(cleaned, pd.DataFrame)
        assert not cleaned.empty

    def test_clean_dataframe_empty(self, mock_analyzer):
        """Test cleaning empty DataFrame."""
        empty_df = pd.DataFrame()
        cleaned = mock_analyzer._clean_dataframe(empty_df)

        assert cleaned.empty

    def test_is_likely_header(self, mock_analyzer):
        """Test header row detection."""
        header_row = pd.Series(['Programa', 'Presupuesto', 'Responsable'])

        # Should recognize as header (nouns, short text)
        is_header = mock_analyzer._is_likely_header(header_row)

        assert isinstance(is_header, bool)

    def test_deduplicate_tables(self, mock_analyzer):
        """Test table deduplication."""
        tables = [
            ExtractedTable(
                df=sample_dataframe,
                page_number=1,
                table_type='financial',
                extraction_method='camelot_lattice',
                confidence_score=0.9
            ),
            ExtractedTable(
                df=sample_dataframe,  # Duplicate
                page_number=1,
                table_type='financial',
                extraction_method='camelot_stream',
                confidence_score=0.85
            )
        ]

        unique = mock_analyzer._deduplicate_tables(tables)

        assert isinstance(unique, list)
        # Should keep only one (highest confidence)
        assert len(unique) <= len(tables)


class TestExtractedTable:
    """Test suite for ExtractedTable data structure."""

    def test_extracted_table_creation(self, sample_dataframe):
        """Test ExtractedTable instantiation."""
        table = ExtractedTable(
            df=sample_dataframe,
            page_number=1,
            table_type='financial',
            extraction_method='camelot_lattice',
            confidence_score=0.9
        )

        assert table.page_number == 1
        assert table.table_type == 'financial'
        assert table.extraction_method == 'camelot_lattice'
        assert table.confidence_score == 0.9
        assert table.is_fragmented is False

    def test_extracted_table_fragmented(self, sample_dataframe):
        """Test fragmented table attributes."""
        table = ExtractedTable(
            df=sample_dataframe,
            page_number=2,
            table_type='financial',
            extraction_method='tabula',
            confidence_score=0.75,
            is_fragmented=True,
            continuation_of=1
        )

        assert table.is_fragmented is True
        assert table.continuation_of == 1


class TestFinancialIndicator:
    """Test suite for FinancialIndicator."""

    def test_financial_indicator_creation(self):
        """Test FinancialIndicator instantiation."""
        indicator = FinancialIndicator(
            source_text='Presupuesto de $500 millones',
            amount=Decimal('500000000'),
            currency='COP',
            fiscal_year=2024,
            funding_source='SGP',
            budget_category='EducaciÃ³n',
            execution_percentage=75.0,
            confidence_interval=(0.7, 0.8),
            risk_level=0.2
        )

        assert indicator.amount == Decimal('500000000')
        assert indicator.currency == 'COP'
        assert indicator.fiscal_year == 2024
        assert indicator.funding_source == 'SGP'
        assert 0.0 <= indicator.risk_level <= 1.0

    def test_financial_indicator_confidence_interval(self):
        """Test confidence interval bounds."""
        indicator = FinancialIndicator(
            source_text='Test',
            amount=Decimal('1000000'),
            currency='COP',
            fiscal_year=2024,
            funding_source='SGP',
            budget_category='Test',
            execution_percentage=None,
            confidence_interval=(0.6, 0.9),
            risk_level=0.1
        )

        assert indicator.confidence_interval[0] < indicator.confidence_interval[1]
        assert 0.0 <= indicator.confidence_interval[0] <= 1.0
        assert 0.0 <= indicator.confidence_interval[1] <= 1.0


class TestResponsibleEntity:
    """Test suite for ResponsibleEntity."""

    def test_responsible_entity_creation(self):
        """Test ResponsibleEntity instantiation."""
        entity = ResponsibleEntity(
            name='SecretarÃ-a de EducaciÃ³n',
            entity_type='secretarÃ-a',
            specificity_score=0.9,
            mentioned_count=5,
            associated_programs=['Programa 1', 'Programa 2'],
            associated_indicators=['Indicador 1'],
            budget_allocated=Decimal('500000000')
        )

        assert entity.name == 'SecretarÃ-a de EducaciÃ³n'
        assert entity.entity_type == 'secretarÃa'
        assert entity.specificity_score == 0.9
        assert len(entity.associated_programs) == 2

    def test_responsible_entity_no_budget(self):
        """Test entity without budget allocation."""
        entity = ResponsibleEntity(
            name='Oficina de PlaneaciÃ³n',
            entity_type='oficina',
            specificity_score=0.8,
            mentioned_count=3,
            associated_programs=[],
            associated_indicators=[],
            budget_allocated=None
        )

        assert entity.budget_allocated is None


class TestQualityScore:
    """Test suite for QualityScore."""

    def test_quality_score_creation(self):
        """Test QualityScore instantiation."""
        score = QualityScore(
            overall_score=0.85,
            financial_feasibility=0.90,
            indicator_quality=0.80,
            responsibility_clarity=0.85,
            temporal_consistency=0.90,
            pdet_alignment=0.75,
            causal_coherence=0.80,
            confidence_interval=(0.80, 0.90),
            evidence={'key': 'value'}
        )

        assert 0.0 <= score.overall_score <= 1.0
        assert 0.0 <= score.financial_feasibility <= 1.0
        assert 0.0 <= score.pdet_alignment <= 1.0
        assert score.confidence_interval[0] < score.confidence_interval[1]

    def test_quality_score_all_dimensions(self):
        """Test all quality score dimensions are bounded [0, 1]."""
        score = QualityScore(
            overall_score=0.75,
            financial_feasibility=0.80,
            indicator_quality=0.70,
            responsibility_clarity=0.85,
            temporal_consistency=0.75,
            pdet_alignment=0.70,
            causal_coherence=0.80,
            confidence_interval=(0.70, 0.80),
            evidence={}
        )

        assert all(0.0 <= getattr(score, field) <= 1.0
                    for field in ['overall_score', 'financial_feasibility',
                                  'indicator_quality', 'responsibility_clarity',
                                  'temporal_consistency', 'pdet_alignment',
                                  'causal_coherence'])


class TestIntegration:
    """Integration tests for financial viability analysis."""

    @pytest.mark.asyncio
    async def test_table_extraction_pipeline(self, mock_analyzer, tmp_path):
        """Test table extraction pipeline with mock PDF."""
        # Create a minimal test - actual PDF extraction requires PDF file
        # This tests the structure without full PDF processing
        assert mock_analyzer is not None

    def test_pdet_alignment_scoring(self):
        """Test PDET alignment scoring logic."""
        context = ColombianMunicipalContext()

        # Verify theory of change structure for scoring
        for pillar in context.PDET_PILLARS:
            assert pillar in context.PDET_THEORY_OF_CHANGE
            toc = context.PDET_THEORY_OF_CHANGE[pillar]
            assert 'outcomes' in toc
            assert 'mediators' in toc
            assert isinstance(toc['lag_years'], int)

    def test_financial_indicator_extraction_pipeline(self):
        """Test financial indicator extraction structure."""
        # Test the data structure requirements
        indicator = FinancialIndicator(
            source_text='SGP de $1,000,000,000 para 2024',
            amount=Decimal('1000000000'),
            currency='COP',
            fiscal_year=2024,
            funding_source='SGP',
            budget_category='InversiÃ³n',
            execution_percentage=None,
            confidence_interval=(0.8, 0.95),
            risk_level=0.15
        )

        assert indicator.amount > 0
        assert indicator.fiscal_year == 2024
        assert indicator.funding_source == 'SGP'



================================================================================
FILE: tests/analysis/test_graph_metrics_fallback.py
================================================================================

"""
Unit tests for Graph Metrics Fallback (graph_metrics_fallback.py)

```

```
Tests graph metrics computation with NetworkX fallback handling,
graceful degradation, and observability integration.
"""
from unittest.mock import patch

import pytest

from farfan_pipeline.core.contracts.runtime_contracts import GraphMetricsInfo
from farfan_pipeline.core.runtime_config import RuntimeConfig
from src.farfan_pipeline.analysis.graph_metrics_fallback import (
    check_networkx_available,
    compute_basic_graph_stats,
    compute_graph_metrics_with_fallback,
)


@pytest.fixture
def sample_edge_list():
    """Create sample edge list for testing."""
    return [
        ('A', 'B'),
        ('B', 'C'),
        ('C', 'D'),
        ('A', 'D')
    ]


@pytest.fixture
def sample_adjacency_dict():
    """Create sample adjacency dictionary."""
    return {
        'A': ['B', 'D'],
        'B': ['C'],
        'C': ['D'],
        'D': []
    }


@pytest.fixture
def runtime_config():
    """Create runtime configuration."""
    return RuntimeConfig(mode='development')


class TestNetworkXAvailability:
    """Test suite for NetworkX availability checking."""

    def test_check_networkx_available_true(self):
        """Test NetworkX availability check when installed."""
        # NetworkX should be available in test environment
        assert check_networkx_available() is True

    @patch('src.farfan_pipeline.analysis.graph_metrics_fallback.check_networkx_available')
    def test_check_networkx_available_false(self, mock_check):
        """Test NetworkX availability check when not installed."""
        mock_check.return_value = False
        assert mock_check() is False


class TestGraphMetricsComputation:
    """Test suite for graph metrics computation with fallback."""

    def test_compute_metrics_with_edge_list(self, sample_edge_list, runtime_config):
        """Test metrics computation with edge list format."""
        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=runtime_config
        )

        assert isinstance(info, GraphMetricsInfo)
        if info.computed:
            assert 'num_nodes' in metrics
            assert 'num_edges' in metrics
            assert metrics['num_nodes'] > 0
            assert metrics['num_edges'] > 0

    def test_compute_metrics_with_adjacency_dict(self, sample_adjacency_dict, runtime_config):
        """Test metrics computation with adjacency dict format."""
        metrics, info = compute_graph_metrics_with_fallback(
            sample_adjacency_dict,
            runtime_config=runtime_config
        )

        assert isinstance(info, GraphMetricsInfo)
        if info.computed:
            assert 'num_nodes' in metrics
            assert 'num_edges' in metrics

    def test_compute_metrics_networkx_available(self, sample_edge_list, runtime_config):
        """Test metrics computation when NetworkX is available."""
        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=runtime_config
        )

        if check_networkx_available():
            assert info.networkx_available is True
            if info.computed:
                assert 'density' in metrics
                assert 'avg_clustering' in metrics
                assert 'num_components' in metrics

    @patch('src.farfan_pipeline.analysis.graph_metrics_fallback.check_networkx_available')
    def test_compute_metrics_networkx_unavailable(self, mock_check, sample_edge_list, runtime_config):
        """Test metrics computation when NetworkX is not available."""
        mock_check.return_value = False

        with patch('src.farfan_pipeline.analysis.graph_metrics_fallback.check_networkx_available', return_value=False):
            metrics, info = compute_graph_metrics_with_fallback(
                sample_edge_list,
                runtime_config=runtime_config
            )

        assert info.networkx_available is False
        assert info.computed is False
        assert info.reason is not None
        assert 'NetworkX not available' in info.reason

    def test_compute_metrics_with_document_id(self, sample_edge_list, runtime_config):
        """Test metrics computation with document ID for logging."""
        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=runtime_config,
            document_id='DOC-001'
        )

        assert isinstance(info, GraphMetricsInfo)

    def test_compute_metrics_invalid_format(self, runtime_config):
        """Test metrics computation with invalid graph format."""
        invalid_data = "invalid graph data"

        metrics, info = compute_graph_metrics_with_fallback(
            invalid_data,
            runtime_config=runtime_config
        )

        # Should handle gracefully
        assert isinstance(info, GraphMetricsInfo)

    def test_compute_metrics_empty_graph(self, runtime_config):
        """Test metrics computation with empty graph."""
        empty_graph = []

        metrics, info = compute_graph_metrics_with_fallback(
            empty_graph,
            runtime_config=runtime_config
        )

        assert isinstance(info, GraphMetricsInfo)


class TestBasicGraphStats:
    """Test suite for basic graph statistics without NetworkX."""

    def test_basic_stats_edge_list(self, sample_edge_list):
        """Test basic stats with edge list format."""
        stats = compute_basic_graph_stats(sample_edge_list)

        assert isinstance(stats, dict)
        assert 'num_nodes' in stats
        assert 'num_edges' in stats
        assert stats['num_nodes'] == 4
        assert stats['num_edges'] == 4
        assert stats['method'] == 'basic_stats_no_networkx'

    def test_basic_stats_adjacency_dict(self, sample_adjacency_dict):
        """Test basic stats with adjacency dict format."""
        stats = compute_basic_graph_stats(sample_adjacency_dict)

        assert isinstance(stats, dict)
        assert 'num_nodes' in stats
        assert 'num_edges' in stats
        assert stats['num_nodes'] == 4
        assert stats['method'] == 'basic_stats_no_networkx'

    def test_basic_stats_invalid_format(self):
        """Test basic stats with invalid format."""
        invalid_data = "invalid"
        stats = compute_basic_graph_stats(invalid_data)

        assert stats['num_nodes'] == 0
        assert stats['num_edges'] == 0
        assert stats['method'] == 'unknown_format'

    def test_basic_stats_empty_edge_list(self):
        """Test basic stats with empty edge list."""
        empty_list = []
        stats = compute_basic_graph_stats(empty_list)

        assert stats['num_nodes'] == 0
        assert stats['num_edges'] == 0


class TestGraphMetricsInfo:
    """Test suite for GraphMetricsInfo structure."""

    def test_graph_metrics_info_computed(self):
        """Test GraphMetricsInfo when metrics are computed."""
        info = GraphMetricsInfo(
            computed=True,
            networkx_available=True,
            reason=None
        )

        assert info.computed is True
        assert info.networkx_available is True
        assert info.reason is None

    def test_graph_metrics_info_not_computed(self):
        """Test GraphMetricsInfo when metrics are not computed."""
        info = GraphMetricsInfo(
            computed=False,
            networkx_available=False,
            reason='NetworkX not available'
        )

        assert info.computed is False
        assert info.networkx_available is False
        assert info.reason == 'NetworkX not available'


class TestFallbackObservability:
    """Test suite for fallback observability integration."""

    @patch('src.farfan_pipeline.analysis.graph_metrics_fallback.log_fallback')
    @patch('src.farfan_pipeline.analysis.graph_metrics_fallback.increment_graph_metrics_skipped')
    @patch('src.farfan_pipeline.analysis.graph_metrics_fallback.check_networkx_available')
    def test_fallback_logging_networkx_unavailable(self, mock_check, mock_increment, mock_log, runtime_config):
        """Test that fallback is logged when NetworkX is unavailable."""
        mock_check.return_value = False

        with patch('src.farfan_pipeline.analysis.graph_metrics_fallback.check_networkx_available', return_value=False):
            metrics, info = compute_graph_metrics_with_fallback(
                [],
                runtime_config=runtime_config,
                document_id='DOC-001'
            )

        # Verify fallback was logged (may or may not be called depending on implementation)
        assert info.computed is False

    def test_runtime_config_default(self, sample_edge_list):
        """Test that default runtime config is used when not provided."""
        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=None
        )

        assert isinstance(info, GraphMetricsInfo)


class TestIntegration:
    """Integration tests for graph metrics with fallback."""

    def test_full_pipeline_with_networkx(self, sample_edge_list, runtime_config):
        """Test full pipeline when NetworkX is available."""
        if not check_networkx_available():
            pytest.skip("NetworkX not available")

        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=runtime_config,
            document_id='TEST-001'
        )

        assert info.networkx_available is True
        if info.computed:
            assert 'num_nodes' in metrics
            assert 'density' in metrics
            assert metrics['num_nodes'] > 0

    def test_graceful_degradation(self, sample_edge_list, runtime_config):
        """Test graceful degradation to basic stats."""
        # Even if NetworkX fails, should return valid GraphMetricsInfo
        metrics, info = compute_graph_metrics_with_fallback(
            sample_edge_list,
            runtime_config=runtime_config
        )

        assert isinstance(metrics, dict)
        assert isinstance(info, GraphMetricsInfo)
        assert info.networkx_available in [True, False]



================================================================================
FILE: tests/analysis/test_recommendation_engine.py
================================================================================

"""
Unit tests for Recommendation Engine (recommendation_engine.py)

Tests rule-based recommendation generation at MICRO, MESO, and MACRO levels,
template rendering, and condition evaluation.
"""
from datetime import datetime

import pytest

from src.farfan_pipeline.analysis.recommendation_engine import (
    Recommendation,
    RecommendationEngine,
    RecommendationSet,
)


@pytest.fixture
def mock_rules():
    """Create mock recommendation rules."""
    return {
        'version': '2.0',
        'rules': [
            {
                'rule_id': 'MICRO-001',
                'level': 'MICRO',
                'when': {
                    'pa_id': 'PA01',
                    'dim_id': 'DIM01',
                    'score_lt': 2.0
                },
                'template': {
                    'problem': 'Problema en {{PAxx}}-{{DIMxx}}',
                    'intervention': 'Implementar mejoras',
                    'indicator': {'name': 'Indicador test'},
                    'responsible': {'entity': 'SecretarÃ-a'},
                    'horizon': {'short': '3 meses'},
                    'verification': ['Verificar resultados']
                }
            },
            {
                'rule_id': 'MESO-001',
                'level': 'MESO',
                'when': {
                    'cluster_id': 'CL01',
                    'score_band': 'BAJO',
                    'variance_level': 'ALTA'
                },
                'template': {
                    'problem': 'Cluster {{cluster_id}} con alto riesgo',
                    'intervention': 'IntervenciÃ³n cluster',
                    'indicator': {'name': 'Indicador cluster'},
                    'responsible': {'entity': 'Equipo'},
                    'horizon': {'medium': '6 meses'},
                    'verification': ['Verificar cluster']
                }
            },
            {
                'rule_id': 'MACRO-001',
                'level': 'MACRO',
                'when': {
                    'overall_score_lt': 65.0
                },
                'template': {
                    'problem': 'Plan con calificaciÃ³n baja',
                    'intervention': 'RevisiÃ³n estratÃ©gica',
                    'indicator': {'name': 'Indicador macro'},
                    'responsible': {'entity': 'AlcaldÃ-a'},
                    'horizon': {'long': '12 meses'},
                    'verification': ['AuditorÃ-a externa']
                }
            }
        ]
    }


@pytest.fixture
def mock_schema():
    """Create mock JSON schema."""
    return {
        'type': 'object',
        'properties': {
            'version': {'type': 'string'},
            'rules': {'type': 'array'}
        },
        'required': ['version', 'rules']
    }


@pytest.fixture
def recommendation_engine(tmp_path, mock_rules, mock_schema):
    """Create recommendation engine with mock files."""
    # Create temporary rules file
    rules_path = tmp_path / 'rules.json'
    schema_path = tmp_path / 'schema.json'

    import json
    with open(rules_path, 'w') as f:
        json.dump(mock_rules, f)
    with open(schema_path, 'w') as f:
        json.dump(mock_schema, f)

    return RecommendationEngine(
        rules_path=str(rules_path),
        schema_path=str(schema_path)
    )


class TestRecommendationEngine:
    """Test suite for RecommendationEngine."""

    def test_initialization(self, recommendation_engine):
        """Test engine initialization."""
        assert recommendation_engine is not None
        assert len(recommendation_engine.rules_by_level['MICRO']) > 0
        assert len(recommendation_engine.rules_by_level['MESO']) > 0
        assert len(recommendation_engine.rules_by_level['MACRO']) > 0

    def test_generate_micro_recommendations_no_match(self, recommendation_engine):
        """Test MICRO recommendations with no matching rules."""
        scores = {
            'PA01-DIM01': 3.0,  # Above threshold
            'PA02-DIM01': 2.5
        }

        result = recommendation_engine.generate_micro_recommendations(scores)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MICRO'
        assert len(result.recommendations) == 0

    def test_generate_micro_recommendations_with_match(self, recommendation_engine):
        """Test MICRO recommendations with matching rule."""
        scores = {
            'PA01-DIM01': 1.5,  # Below threshold of 2.0
            'PA02-DIM01': 2.5
        }

        result = recommendation_engine.generate_micro_recommendations(scores)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MICRO'
        assert len(result.recommendations) > 0
        assert result.recommendations[0].rule_id == 'MICRO-001'

    def test_micro_recommendation_metadata(self, recommendation_engine):
        """Test MICRO recommendation includes metadata."""
        scores = {'PA01-DIM01': 1.0}

        result = recommendation_engine.generate_micro_recommendations(scores)

        rec = result.recommendations[0]
        assert 'score_key' in rec.metadata
        assert 'actual_score' in rec.metadata
        assert 'threshold' in rec.metadata
        assert rec.metadata['score_key'] == 'PA01-DIM01'

    def test_generate_meso_recommendations_no_match(self, recommendation_engine):
        """Test MESO recommendations with no matching rules."""
        cluster_data = {
            'CL01': {
                'score': 75.0,  # ALTO
                'variance': 0.05,  # BAJA
                'weak_pa': 'PA01'
            }
        }

        result = recommendation_engine.generate_meso_recommendations(cluster_data)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MESO'

    def test_generate_meso_recommendations_with_match(self, recommendation_engine):
        """Test MESO recommendations with matching rule."""
        cluster_data = {
            'CL01': {
                'score': 50.0,  # BAJO
                'variance': 0.25,  # ALTA
                'weak_pa': 'PA02'
            }
        }

        result = recommendation_engine.generate_meso_recommendations(cluster_data)

        assert isinstance(result, RecommendationSet)
        assert len(result.recommendations) > 0

    def test_generate_macro_recommendations(self, recommendation_engine):
        """Test MACRO recommendations generation."""
        plan_metrics = {
            'overall_score': 60.0,  # Below threshold
            'coverage': 0.75,
            'coherence': 0.80
        }

        result = recommendation_engine.generate_macro_recommendations(plan_metrics)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MACRO'

    def test_template_rendering_micro(self, recommendation_engine):
        """Test MICRO template variable substitution."""
        template = {
            'problem': 'Problema en {{PAxx}}-{{DIMxx}}',
            'intervention': 'IntervenciÃ³n',
            'indicator': {},
            'responsible': {},
            'horizon': {},
            'verification': []
        }

        rendered = recommendation_engine._render_micro_template(
            template, 'PA01', 'DIM02', {}
        )

        assert 'PA01' in rendered['problem']
        assert 'DIM02' in rendered['problem']

    def test_reload_rules(self, recommendation_engine):
        """Test hot-reloading of rules."""
        initial_count = len(recommendation_engine.rules_by_level['MICRO'])

        # Reload should work without error
        recommendation_engine.reload_rules()

        reloaded_count = len(recommendation_engine.rules_by_level['MICRO'])
        assert reloaded_count == initial_count


class TestRecommendation:
    """Test suite for Recommendation data structure."""

    def test_recommendation_creation(self):
        """Test recommendation creation."""
        rec = Recommendation(
            rule_id='TEST-001',
            level='MICRO',
            problem='Test problem',
            intervention='Test intervention',
            indicator={'name': 'Test indicator'},
            responsible={'entity': 'Test entity'},
            horizon={'short': '3 months'},
            verification=['Test verification']
        )

        assert rec.rule_id == 'TEST-001'
        assert rec.level == 'MICRO'
        assert rec.problem == 'Test problem'

    def test_recommendation_to_dict(self):
        """Test recommendation serialization."""
        rec = Recommendation(
            rule_id='TEST-001',
            level='MICRO',
            problem='Test problem',
            intervention='Test intervention',
            indicator={'name': 'Test indicator'},
            responsible={'entity': 'Test entity'},
            horizon={'short': '3 months'},
            verification=['Test verification']
        )

        rec_dict = rec.to_dict()

        assert isinstance(rec_dict, dict)
        assert rec_dict['rule_id'] == 'TEST-001'
        assert rec_dict['level'] == 'MICRO'

    def test_recommendation_enhanced_fields(self):
        """Test recommendation with v2.0 enhanced fields."""
        rec = Recommendation(
            rule_id='TEST-001',
            level='MICRO',
            problem='Test',
            intervention='Test',
            indicator={},
            responsible={},
            horizon={},
            verification=[],
            execution={'steps': ['step1', 'step2']},
            budget={'estimated': 1000000}
        )

        assert rec.execution is not None
        assert rec.budget is not None
        assert rec.execution['steps'][0] == 'step1'


class TestRecommendationSet:
    """Test suite for RecommendationSet."""

    def test_recommendation_set_creation(self):
        """Test recommendation set creation."""
        rec_set = RecommendationSet(
            level='MICRO',
            recommendations=[],
            generated_at=datetime.now().isoformat(),
            total_rules_evaluated=10,
            rules_matched=3
        )

        assert rec_set.level == 'MICRO'
        assert rec_set.total_rules_evaluated == 10
        assert rec_set.rules_matched == 3

    def test_recommendation_set_to_dict(self):
        """Test recommendation set serialization."""
        rec = Recommendation(
            rule_id='TEST-001',
            level='MICRO',
            problem='Test',
            intervention='Test',
            indicator={},
            responsible={},
            horizon={},
            verification=[]
        )

        rec_set = RecommendationSet(
            level='MICRO',
            recommendations=[rec],
            generated_at=datetime.now().isoformat(),
            total_rules_evaluated=5,
            rules_matched=1
        )

        rec_set_dict = rec_set.to_dict()

        assert isinstance(rec_set_dict, dict)
        assert rec_set_dict['level'] == 'MICRO'
        assert len(rec_set_dict['recommendations']) == 1


class TestConditionEvaluation:
    """Test suite for condition evaluation logic."""

    def test_check_meso_conditions_score_band(self, recommendation_engine):
        """Test MESO score band condition checking."""
        # BAJO band: score < 55
        assert recommendation_engine._check_meso_conditions(
            score=50.0, variance=0.1, weak_pa='PA01',
            score_band='BAJO', variance_level='MEDIA',
            variance_threshold=None, weak_pa_id=None
        )

        # Should fail for score >= 55
        assert not recommendation_engine._check_meso_conditions(
            score=60.0, variance=0.1, weak_pa='PA01',
            score_band='BAJO', variance_level='MEDIA',
            variance_threshold=None, weak_pa_id=None
        )

    def test_check_meso_conditions_variance(self, recommendation_engine):
        """Test MESO variance level condition checking."""
        # ALTA variance: variance >= 0.18
        assert recommendation_engine._check_meso_conditions(
            score=50.0, variance=0.20, weak_pa='PA01',
            score_band='BAJO', variance_level='ALTA',
            variance_threshold=18.0, weak_pa_id=None
        )


class TestIntegration:
    """Integration tests for recommendation engine."""

    def test_full_pipeline_micro(self, recommendation_engine):
        """Test full MICRO recommendation pipeline."""
        scores = {
            'PA01-DIM01': 1.5,
            'PA02-DIM01': 2.5,
            'PA03-DIM02': 1.0
        }

        result = recommendation_engine.generate_micro_recommendations(scores)

        assert isinstance(result, RecommendationSet)
        assert result.generated_at is not None
        assert result.total_rules_evaluated > 0

    def test_full_pipeline_meso(self, recommendation_engine):
        """Test full MESO recommendation pipeline."""
        cluster_data = {
            'CL01': {'score': 45.0, 'variance': 0.25, 'weak_pa': 'PA01'},
            'CL02': {'score': 75.0, 'variance': 0.05, 'weak_pa': 'PA02'}
        }

        result = recommendation_engine.generate_meso_recommendations(cluster_data)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MESO'

    def test_full_pipeline_macro(self, recommendation_engine):
        """Test full MACRO recommendation pipeline."""
        plan_metrics = {
            'overall_score': 60.0,
            'coverage': 0.70,
            'coherence': 0.75
        }

        result = recommendation_engine.generate_macro_recommendations(plan_metrics)

        assert isinstance(result, RecommendationSet)
        assert result.level == 'MACRO'



================================================================================
FILE: tests/analysis/test_semantic_analyzer.py
================================================================================

"""
Unit tests for SemanticAnalyzer (Analyzer_one.py)

Tests semantic cube extraction, value chain classification,
policy domain mapping, and cross-cutting theme detection.
"""

import numpy as np
import pytest

from src.farfan_pipeline.analysis.Analyzer_one import (
    MunicipalOntology,
    SemanticAnalyzer,
    ValueChainLink,
)


@pytest.fixture
def ontology():
    """Create municipal ontology for testing."""
    return MunicipalOntology()


@pytest.fixture
def semantic_analyzer(ontology):
    """Create semantic analyzer with test ontology."""
    return SemanticAnalyzer(
        ontology=ontology,
        max_features=100,
        ngram_range=(1, 2),
        similarity_threshold=0.3
    )


@pytest.fixture
def sample_segments():
    """Sample document segments for testing."""
    return [
        "El diagnÃ³stico territorial identificÃ³ las principales necesidades de la poblaciÃ³n",
        "La estrategia de desarrollo econÃ³mico incluye competitividad y emprendimiento",
        "La implementaciÃ³n de servicios mejorarÃ¡ las condiciones de vida",
        "Transparencia y participaciÃ³n ciudadana son temas transversales"
    ]


class TestSemanticAnalyzer:
    """Test suite for SemanticAnalyzer."""

    def test_initialization(self, semantic_analyzer, ontology):
        """Test semantic analyzer initialization."""
        assert semantic_analyzer.ontology == ontology
        assert semantic_analyzer.max_features == 100
        assert semantic_analyzer.ngram_range == (1, 2)
        assert semantic_analyzer.similarity_threshold == 0.3

    def test_extract_semantic_cube_empty_segments(self, semantic_analyzer):
        """Test semantic cube extraction with empty segments."""
        result = semantic_analyzer.extract_semantic_cube([])

        assert result["dimensions"]["value_chain_links"] == {}
        assert result["dimensions"]["policy_domains"] == {}
        assert result["measures"]["overall_coherence"] == 0.0
        assert result["metadata"]["total_segments"] == 0

    def test_extract_semantic_cube_valid_segments(self, semantic_analyzer, sample_segments):
        """Test semantic cube extraction with valid segments."""
        result = semantic_analyzer.extract_semantic_cube(sample_segments)

        assert "dimensions" in result
        assert "measures" in result
        assert "metadata" in result
        assert result["metadata"]["total_segments"] == len(sample_segments)
        assert isinstance(result["measures"]["overall_coherence"], float)
        assert 0.0 <= result["measures"]["overall_coherence"] <= 1.0

    def test_classify_value_chain_link(self, semantic_analyzer):
        """Test value chain link classification."""
        segment = "diagnÃ³stico territorial con mapeo de actores y evaluaciÃ³n de necesidades"
        scores = semantic_analyzer._classify_value_chain_link(segment)

        assert isinstance(scores, dict)
        assert "diagnostic_planning" in scores
        assert all(0.0 <= score <= 1.0 for score in scores.values())
        assert scores["diagnostic_planning"] > 0.0

    def test_classify_policy_domain(self, semantic_analyzer):
        """Test policy domain classification."""
        segment = "desarrollo econÃ³mico con Ã©nfasis en competitividad y empleo"
        scores = semantic_analyzer._classify_policy_domain(segment)

        assert isinstance(scores, dict)
        assert "economic_development" in scores
        assert all(0.0 <= score <= 1.0 for score in scores.values())
        assert scores["economic_development"] > 0.0

    def test_classify_cross_cutting_themes(self, semantic_analyzer):
        """Test cross-cutting theme classification."""
        segment = "transparencia, rendiciÃ³n de cuentas y participaciÃ³n ciudadana"
        scores = semantic_analyzer._classify_cross_cutting_themes(segment)

        assert isinstance(scores, dict)
        assert "governance" in scores
        assert all(0.0 <= score <= 1.0 for score in scores.values())
        assert scores["governance"] > 0.0

    def test_vectorize_segments(self, semantic_analyzer, sample_segments):
        """Test segment vectorization."""
        vectors = semantic_analyzer._vectorize_segments(sample_segments)

        assert isinstance(vectors, (np.ndarray, list))
        if isinstance(vectors, np.ndarray):
            assert vectors.shape[0] == len(sample_segments)
            assert vectors.shape[1] > 0

    def test_process_segment(self, semantic_analyzer):
        """Test individual segment processing."""
        segment = "Esta es una oraciÃ³n de prueba con varias palabras."
        vector = np.zeros(100)

        result = semantic_analyzer._process_segment(segment, 0, vector)

        assert result["segment_id"] == 0
        assert result["text"] == segment
        assert "word_count" in result
        assert "sentence_count" in result
        assert "semantic_density" in result
        assert "coherence_score" in result
        assert 0.0 <= result["semantic_density"] <= 1.0
        assert 0.0 <= result["coherence_score"] <= 1.0

    def test_calculate_semantic_complexity(self, semantic_analyzer, sample_segments):
        """Test semantic complexity calculation."""
        cube = semantic_analyzer.extract_semantic_cube(sample_segments)
        complexity = semantic_analyzer._calculate_semantic_complexity(cube)

        assert isinstance(complexity, float)
        assert complexity >= 0.0

    def test_similarity_threshold_filtering(self, semantic_analyzer):
        """Test that similarity threshold filters classifications."""
        # High threshold should filter out weak matches
        analyzer_strict = SemanticAnalyzer(
            semantic_analyzer.ontology,
            similarity_threshold=0.9
        )

        segment = "texto genÃ©rico sin palabras clave especÃ-ficas"
        cube = analyzer_strict.extract_semantic_cube([segment])

        # Should have few or no classifications with high threshold
        total_classifications = sum(
            len(items) for items in cube["dimensions"]["value_chain_links"].values()
        )
        assert total_classifications >= 0

    def test_empty_semantic_cube_structure(self, semantic_analyzer):
        """Test empty semantic cube has correct structure."""
        cube = semantic_analyzer._empty_semantic_cube()

        assert "dimensions" in cube
        assert "measures" in cube
        assert "metadata" in cube
        assert cube["dimensions"]["value_chain_links"] == {}
        assert cube["measures"]["overall_coherence"] == 0.0
        assert cube["measures"]["semantic_complexity"] == 0.0


class TestMunicipalOntology:
    """Test suite for MunicipalOntology."""

    def test_value_chain_links_structure(self, ontology):
        """Test value chain links are properly structured."""
        assert "diagnostic_planning" in ontology.value_chain_links
        assert "strategic_planning" in ontology.value_chain_links
        assert "implementation" in ontology.value_chain_links

        for link_name, link in ontology.value_chain_links.items():
            assert isinstance(link, ValueChainLink)
            assert link.name == link_name
            assert isinstance(link.instruments, list)
            assert isinstance(link.mediators, list)
            assert isinstance(link.outputs, list)
            assert isinstance(link.outcomes, list)
            assert link.lead_time_days > 0
            assert isinstance(link.conversion_rates, dict)

    def test_policy_domains_coverage(self, ontology):
        """Test policy domains are comprehensive."""
        assert "economic_development" in ontology.policy_domains
        assert "social_development" in ontology.policy_domains
        assert "territorial_development" in ontology.policy_domains
        assert "institutional_development" in ontology.policy_domains

        for domain, keywords in ontology.policy_domains.items():
            assert isinstance(keywords, list)
            assert len(keywords) > 0

    def test_cross_cutting_themes(self, ontology):
        """Test cross-cutting themes are defined."""
        assert "governance" in ontology.cross_cutting_themes
        assert "equity" in ontology.cross_cutting_themes
        assert "sustainability" in ontology.cross_cutting_themes
        assert "innovation" in ontology.cross_cutting_themes

        for theme, keywords in ontology.cross_cutting_themes.items():
            assert isinstance(keywords, list)
            assert len(keywords) > 0


class TestSemanticCubeIntegration:
    """Integration tests for semantic cube extraction."""

    def test_full_pipeline_with_realistic_segments(self, semantic_analyzer):
        """Test full semantic cube extraction pipeline."""
        segments = [
            "El diagnÃ³stico participativo identificÃ³ brechas en educaciÃ³n y salud",
            "La planificaciÃ³n estratÃ©gica prioriza el desarrollo econÃ³mico local",
            "La implementaciÃ³n incluye proyectos de infraestructura vial",
            "La sostenibilidad ambiental y la equidad de gÃ©nero son ejes transversales"
        ]

        cube = semantic_analyzer.extract_semantic_cube(segments)

        # Validate structure
        assert cube["metadata"]["total_segments"] == 4
        assert len(cube["measures"]["semantic_density"]) == 4
        assert len(cube["measures"]["coherence_scores"]) == 4

        # Validate dimensions have content
        assert len(cube["dimensions"]["value_chain_links"]) > 0
        assert len(cube["dimensions"]["policy_domains"]) > 0

        # Validate measures are computed
        assert isinstance(cube["measures"]["overall_coherence"], float)
        assert isinstance(cube["measures"]["semantic_complexity"], float)

    def test_semantic_cube_with_multilingual_content(self, semantic_analyzer):
        """Test robustness with mixed language content."""
        segments = [
            "DiagnÃ³stico with some English words territorial",
            "Strategic planning para el desarrollo"
        ]

        # Should not crash with mixed content
        cube = semantic_analyzer.extract_semantic_cube(segments)
        assert cube["metadata"]["total_segments"] == 2



================================================================================
FILE: tests/analysis/test_teoria_cambio.py
================================================================================

"""
Unit tests for Teoria Cambio (teoria_cambio.py)

Tests theory of change validation, DAG construction,
Monte Carlo simulation, and axiom enforcement.
"""

import networkx as nx
import pytest

from farfan_pipeline.core.types import CategoriaCausal
from src.farfan_pipeline.analysis.teoria_cambio import (
    AdvancedDAGValidator,
    AdvancedGraphNode,
    MonteCarloAdvancedResult,
    TeoriaCambio,
    ValidacionResultado,
    _create_advanced_seed,
)


@pytest.fixture
def teoria_cambio():
    """Create TeoriaCambio instance for testing."""
    return TeoriaCambio()


@pytest.fixture
def sample_dag():
    """Create sample DAG for testing."""
    G = nx.DiGraph()
    G.add_node('A', categoria=CategoriaCausal.INSUMOS, nivel=1)
    G.add_node('B', categoria=CategoriaCausal.PROCESOS, nivel=2)
    G.add_node('C', categoria=CategoriaCausal.PRODUCTOS, nivel=3)
    G.add_node('D', categoria=CategoriaCausal.RESULTADOS, nivel=4)
    G.add_node('E', categoria=CategoriaCausal.CAUSALIDAD, nivel=5)

    G.add_edge('A', 'B', peso=1.0)
    G.add_edge('B', 'C', peso=1.0)
    G.add_edge('C', 'D', peso=1.0)
    G.add_edge('D', 'E', peso=1.0)

    return G


class TestTeoriaCambio:
    """Test suite for TeoriaCambio theory of change validation."""

    def test_initialization(self, teoria_cambio):
        """Test TeoriaCambio initialization."""
        assert teoria_cambio is not None
        assert teoria_cambio._grafo_cache is None
        assert teoria_cambio._cache_valido is False

    def test_construir_grafo_causal(self, teoria_cambio):
        """Test canonical causal graph construction."""
        grafo = teoria_cambio.construir_grafo_causal()

        assert isinstance(grafo, nx.DiGraph)
        assert grafo.number_of_nodes() == len(CategoriaCausal)
        assert grafo.number_of_edges() > 0

        # Verify all categories are present
        node_names = set(grafo.nodes())
        expected_names = {cat.name for cat in CategoriaCausal}
        assert node_names == expected_names

    def test_grafo_causal_caching(self, teoria_cambio):
        """Test that causal graph is cached after first construction."""
        grafo1 = teoria_cambio.construir_grafo_causal()
        grafo2 = teoria_cambio.construir_grafo_causal()

        # Should return same cached instance
        assert grafo1 is grafo2
        assert teoria_cambio._cache_valido is True

    def test_es_conexion_valida(self):
        """Test valid connection checking between categories."""
        # INSUMOS -> PROCESOS should be valid
        assert TeoriaCambio._es_conexion_valida(
            CategoriaCausal.INSUMOS,
            CategoriaCausal.PROCESOS
        )

        # INSUMOS -> CAUSALIDAD should be invalid (skipping levels)
        assert not TeoriaCambio._es_conexion_valida(
            CategoriaCausal.INSUMOS,
            CategoriaCausal.CAUSALIDAD
        )

    def test_validacion_completa_valid_graph(self, teoria_cambio, sample_dag):
        """Test complete validation with valid graph."""
        resultado = teoria_cambio.validacion_completa(sample_dag)

        assert isinstance(resultado, ValidacionResultado)
        assert resultado.es_valida is True
        assert len(resultado.violaciones_orden) == 0
        assert len(resultado.categorias_faltantes) == 0
        assert len(resultado.caminos_completos) > 0

    def test_validacion_completa_missing_category(self, teoria_cambio):
        """Test validation with missing category."""
        # Create incomplete DAG missing CAUSALIDAD
        G = nx.DiGraph()
        G.add_node('A', categoria=CategoriaCausal.INSUMOS, nivel=1)
        G.add_node('B', categoria=CategoriaCausal.PROCESOS, nivel=2)
        G.add_edge('A', 'B', peso=1.0)

        resultado = teoria_cambio.validacion_completa(G)

        assert resultado.es_valida is False
        assert len(resultado.categorias_faltantes) > 0
        assert CategoriaCausal.CAUSALIDAD in resultado.categorias_faltantes

    def test_validacion_completa_orden_violation(self, teoria_cambio):
        """Test validation with causal order violation."""
        # Create DAG with backward connection
        G = nx.DiGraph()
        G.add_node('A', categoria=CategoriaCausal.PROCESOS, nivel=2)
        G.add_node('B', categoria=CategoriaCausal.INSUMOS, nivel=1)
        G.add_edge('A', 'B', peso=1.0)  # Backward connection

        resultado = teoria_cambio.validacion_completa(G)

        assert len(resultado.violaciones_orden) > 0

    def test_extraer_categorias(self, teoria_cambio, sample_dag):
        """Test category extraction from graph."""
        categorias = teoria_cambio._extraer_categorias(sample_dag)

        assert isinstance(categorias, set)
        assert 'INSUMOS' in categorias
        assert 'CAUSALIDAD' in categorias

    def test_validar_orden_causal(self, teoria_cambio, sample_dag):
        """Test causal order validation."""
        violaciones = teoria_cambio._validar_orden_causal(sample_dag)

        assert isinstance(violaciones, list)
        assert len(violaciones) == 0  # sample_dag should have no violations

    def test_encontrar_caminos_completos(self, teoria_cambio, sample_dag):
        """Test finding complete paths from INSUMOS to CAUSALIDAD."""
        caminos = teoria_cambio._encontrar_caminos_completos(sample_dag)

        assert isinstance(caminos, list)
        assert len(caminos) > 0

        # Verify path goes from INSUMOS to CAUSALIDAD
        for camino in caminos:
            first_node = sample_dag.nodes[camino[0]]
            last_node = sample_dag.nodes[camino[-1]]
            assert first_node['categoria'] == CategoriaCausal.INSUMOS
            assert last_node['categoria'] == CategoriaCausal.CAUSALIDAD

    def test_generar_sugerencias(self, teoria_cambio):
        """Test suggestion generation."""
        resultado_invalido = ValidacionResultado(
            es_valida=False,
            categorias_faltantes=[CategoriaCausal.CAUSALIDAD],
            violaciones_orden=[('A', 'B')],
            caminos_completos=[]
        )

        sugerencias = teoria_cambio._generar_sugerencias_internas(resultado_invalido)

        assert isinstance(sugerencias, list)
        assert len(sugerencias) > 0
        assert any('CAUSALIDAD' in s for s in sugerencias)


class TestAdvancedGraphNode:
    """Test suite for AdvancedGraphNode."""

    def test_node_creation(self):
        """Test node creation with basic attributes."""
        node = AdvancedGraphNode(
            name='TestNode',
            dependencies={'dep1', 'dep2'},
            metadata={'key': 'value'},
            role='variable'
        )

        assert node.name == 'TestNode'
        assert 'dep1' in node.dependencies
        assert node.metadata['key'] == 'value'
        assert node.role == 'variable'

    def test_node_metadata_normalization(self):
        """Test metadata normalization with defaults."""
        node = AdvancedGraphNode(
            name='TestNode',
            metadata={}
        )

        assert 'created' in node.metadata
        assert 'confidence' in node.metadata
        assert 0.0 <= node.metadata['confidence'] <= 1.0

    def test_node_invalid_role(self):
        """Test that invalid role raises error."""
        with pytest.raises(ValueError, match='Invalid role'):
            AdvancedGraphNode(
                name='TestNode',
                role='invalid_role'
            )

    def test_node_empty_name(self):
        """Test that empty name raises error."""
        with pytest.raises(ValueError, match='non-empty string'):
            AdvancedGraphNode(name='')

    def test_node_serialization(self):
        """Test node serialization to dictionary."""
        node = AdvancedGraphNode(
            name='TestNode',
            dependencies={'dep1'},
            role='produto'
        )

        serialized = node.to_serializable_dict()

        assert isinstance(serialized, dict)
        assert serialized['name'] == 'TestNode'
        assert 'dep1' in serialized['dependencies']
        assert serialized['role'] == 'produto'


class TestDeterministicSeeding:
    """Test suite for deterministic seed generation."""

    def test_create_advanced_seed_deterministic(self):
        """Test seed generation is deterministic."""
        seed1 = _create_advanced_seed('plan_name', 'salt')
        seed2 = _create_advanced_seed('plan_name', 'salt')

        assert seed1 == seed2

    def test_create_advanced_seed_different_inputs(self):
        """Test different inputs produce different seeds."""
        seed1 = _create_advanced_seed('plan1', 'salt')
        seed2 = _create_advanced_seed('plan2', 'salt')

        assert seed1 != seed2

    def test_create_advanced_seed_salt_effect(self):
        """Test salt parameter affects seed."""
        seed1 = _create_advanced_seed('plan', 'salt1')
        seed2 = _create_advanced_seed('plan', 'salt2')

        assert seed1 != seed2

    def test_seed_is_valid_integer(self):
        """Test seed is valid 64-bit integer."""
        seed = _create_advanced_seed('test_plan', '')

        assert isinstance(seed, int)
        assert seed >= 0
        assert seed < 2**64


class TestAdvancedDAGValidator:
    """Test suite for Advanced DAG validator with Monte Carlo."""

    @pytest.fixture
    def validator(self):
        """Create DAG validator for testing."""
        return AdvancedDAGValidator(iterations=100)

    def test_validator_initialization(self, validator):
        """Test validator initialization."""
        assert validator is not None
        assert hasattr(validator, 'iterations')

    def test_validate_dag_acyclic(self, validator, sample_dag):
        """Test validation of acyclic DAG."""
        # This would test the full validation
        assert nx.is_directed_acyclic_graph(sample_dag)

```

```python
    def test_monte_carlo_simulation_structure(self):
        """Test Monte Carlo result structure."""
        result = MonteCarloAdvancedResult(
            plan_name='Test Plan',
            seed=42,
            timestamp='2024-01-01T00:00:00',
            total_iterations=1000,
            acyclic_count=950,
            p_value=0.95,
            bayesian_posterior=0.92,
            confidence_interval=(0.88, 0.96),
            statistical_power=0.85,
            edge_sensitivity={'edge1': 0.8},
            node_importance={'node1': 0.9},
            robustness_score=0.87,
            reproducible=True,
            convergence_achieved=True,
            adequate_power=True,
            computation_time=1.5,
            graph_statistics={'nodes': 5, 'edges': 4},
            test_parameters={'alpha': 0.05}
        )

        assert result.plan_name == 'Test Plan'
        assert result.seed == 42
        assert result.reproducible is True
        assert result.p_value == 0.95
        assert 0.0 <= result.bayesian_posterior <= 1.0


```