src/orchestration/method_registry.py

```python
"""Method Registry with lazy instantiation and injection pattern.

This module implements a method injection factory that:
1. Loads only the methods needed (not full classes)
2. Instantiates classes lazily (only when first method is called)
3. Caches instances for reuse with TTL and memory management
4. Isolates errors per method (failures don't cascade)
5. Allows direct function injection (bypassing classes)

Architecture:
    MethodRegistry
        ?? _class_paths: mapping of class names to import paths
        ?? _instance_cache: lazily instantiated class instances with TTL
        ?? _direct_methods: directly injected functions
        ?? get_method(): returns callable for (class_name, method_name)

Memory Management:
- TTL-based eviction for instance cache entries
- Weakref support for garbage collection
- Explicit cache clearing between pipeline runs
- Memory profiling hooks for observability

Benefits:
- No upfront class loading (lightweight imports)
- Failed classes don't block working methods
- Direct function injection for custom implementations
- Instance reuse through caching with memory bounds
- Prevents memory bloat in long-lived processes
"""
from __future__ import annotations

import logging
import threading
import time
import weakref
from dataclasses import dataclass
from importlib import import_module
from typing import Any, Callable

logger = logging.getLogger(__name__)


class MethodRegistryError(RuntimeError):
    """Raised when a method cannot be retrieved."""


@dataclass
class CacheEntry:
    """Cache entry with TTL tracking."""

    instance: Any
    created_at: float
    last_accessed: float
    access_count: int = 0

    def is_expired(self, ttl_seconds: float) -> bool:
        """Check if entry has exceeded TTL."""
        if ttl_seconds <= 0:
            return False
        return (time.time() - self.last_accessed) > ttl_seconds

    def touch(self) -> None:
        """Update access timestamp and counter."""
        self.last_accessed = time.time()
        self.access_count += 1


class MethodRegistry:
    """Registry for lazy method injection without full class instantiation.

    Features memory management with TTL-based eviction and weakref support.
    """

    def __init__(
        self,
        class_paths: dict[str, str] | None = None,
        cache_ttl_seconds: float = 300.0,
        enable_weakref: bool = False,
        max_cache_size: int = 100,
    ) -> None:
        """Initialize the method registry.

        Args:
            class_paths: Optional mapping of class names to import paths.
                        If None, uses default paths from class_registry.
            cache_ttl_seconds: Time-to-live for cache entries in seconds.
                             Set to 0 to disable TTL-based eviction.
            enable_weakref: If True, use weak references for instances.
            max_cache_size: Maximum number of instances to cache.
        """
        # Import class paths from existing registry
        if class_paths is None:
            from orchestration.class_registry import get_class_paths
            class_paths = dict(get_class_paths())

        self._class_paths = class_paths
        self._cache: dict[str, CacheEntry] = {}
        self._weakref_cache: dict[str, weakref.ref[Any]] = {}
        self._direct_methods: dict[tuple[str, str], Callable[..., Any]] = {}
        self._failed_classes: set[str] = set()
        self._lock = threading.Lock()
        self._cache_ttl_seconds = cache_ttl_seconds
        self._enable_weakref = enable_weakref
        self._max_cache_size = max_cache_size

        # Special instantiation rules (from original MethodExecutor)
        self._special_instantiation: dict[str, Callable[[type], Any]] = {}

        # Metrics
        self._cache_hits = 0
        self._cache_misses = 0
        self._evictions = 0
        self._total_instantiations = 0

    def inject_method(
        self,
        class_name: str,
        method_name: str,
        method: Callable[..., Any],
    ) -> None:
        """Directly inject a method without needing a class.

        This allows bypassing class instantiation entirely.

        Args:
            class_name: Virtual class name for routing
            method_name: Method name
            method: Callable to inject
        """
        key = (class_name, method_name)
        self._direct_methods[key] = method
        logger.info(
            "method_injected_directly",
            class_name=class_name,
            method_name=method_name,
        )

    def register_instantiation_rule(
        self,
        class_name: str,
        instantiator: Callable[[type], Any],
    ) -> None:
        """Register special instantiation logic for a class.

        Args:
            class_name: Class name requiring special instantiation
            instantiator: Function that takes class type and returns instance
        """
        self._special_instantiation[class_name] = instantiator
        logger.debug(
            "instantiation_rule_registered",
            class_name=class_name,
        )

    def _load_class(self, class_name: str) -> type:
        """Load a class type from import path.

        Args:
            class_name: Name of class to load
        Returns:
            Class type

        Raises:
            MethodRegistryError: If class cannot be loaded
        """
        if class_name not in self._class_paths:
            raise MethodRegistryError(
                f"Class '{class_name}' not found in registry paths"
            )

        path = self._class_paths[class_name]
        module_name, _, attr_name = path.rpartition(".")

        if not module_name:
            raise MethodRegistryError(
                f"Invalid path for '{class_name}': {path}"
            )

        try:
            module = import_module(module_name)
            cls = getattr(module, attr_name)

            if not isinstance(cls, type):
                raise MethodRegistryError(
                    f"'{class_name}' is not a class: {type(cls).__name__}"
                )

            return cls

        except ImportError as exc:
            raise MethodRegistryError(
                f"Cannot import class '{class_name}' from {path}: {exc}"
            ) from exc
        except AttributeError as exc:
            raise MethodRegistryError(
                f"Class '{attr_name}' not found in module {module_name}: {exc}"
            ) from exc

    def _instantiate_class(self, class_name: str, cls: type) -> Any:
        """Instantiate a class using special rules or default constructor.

        Args:
            class_name: Name of class (for special rule lookup)
            cls: Class type to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If instantiation fails
        """
        # Use special instantiation rule if registered
        if class_name in self._special_instantiation:
            try:
                instantiator = self._special_instantiation[class_name]
                instance = instantiator(cls)
                logger.debug(
                    "class_instantiated_with_special_rule",
                    class_name=class_name,
                )
                return instance
            except Exception as exc:
                raise MethodRegistryError(
                    f"Special instantiation failed for '{class_name}': {exc}"
                ) from exc

        # Default instantiation (no-args constructor)
        try:
            instance = cls()
            logger.debug(
                "class_instantiated_default",
                class_name=class_name,
            )
            return instance
        except Exception as exc:
            raise MethodRegistryError(
                f"Default instantiation failed for '{class_name}': {exc}"
            ) from exc

    def _get_instance(self, class_name: str) -> Any:
        """Get or create instance of a class (lazy + cached).

        Args:
            class_name: Name of class to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If class cannot be instantiated
        """
        # Check if already failed
        if class_name in self._failed_classes:
            raise MethodRegistryError(
                f"Class '{class_name}' previously failed to instantiate"
            )

        # Use a lock to ensure thread-safe instantiation
        with self._lock:
            # Check weakref cache first
            if self._enable_weakref and class_name in self._weakref_cache:
                instance = self._weakref_cache[class_name]()
                if instance is not None:
                    self._cache_hits += 1
                    logger.debug(
                        "class_retrieved_from_weakref_cache",
                        class_name=class_name,
                    )
                    return instance
                else:
                    # Weakref was garbage collected
                    del self._weakref_cache[class_name]

            # Check regular cache and evict if expired
            if class_name in self._cache:
                entry = self._cache[class_name]
                if entry.is_expired(self._cache_ttl_seconds):
                    logger.info(
                        "cache_entry_expired",
                        class_name=class_name,
                        age_seconds=time.time() - entry.created_at,
                        access_count=entry.access_count,
                    )
                    del self._cache[class_name]
                    self._evictions += 1
                else:
                    entry.touch()
                    self._cache_hits += 1
                    return entry.instance

            # Cache miss - need to instantiate
            self._cache_misses += 1

            # Evict oldest entries if cache is full
            self._evict_if_full()

            # Load and instantiate class
            try:
                cls = self._load_class(class_name)
                instance = self._instantiate_class(class_name, cls)
                self._total_instantiations += 1

                # Store in appropriate cache
                if self._enable_weakref:
                    self._weakref_cache[class_name] = weakref.ref(instance)
                    logger.info(
                        "class_instantiated_weakref",
                        class_name=class_name,
                    )
                else:
                    entry = CacheEntry(
                        instance=instance,
                        created_at=time.time(),
                        last_accessed=time.time(),
                        access_count=1,
                    )
                    self._cache[class_name] = entry
                    logger.info(
                        "class_instantiated_cached",
                        class_name=class_name,
                    )
                return instance

            except MethodRegistryError:
                # Mark as failed to avoid repeated attempts
                self._failed_classes.add(class_name)
                raise

    def _evict_if_full(self) -> None:
        """Evict oldest cache entries if cache size exceeds maximum."""
        if len(self._cache) <= self._max_cache_size:
            return

        # Sort by last accessed time and evict oldest
        sorted_entries = sorted(
            self._cache.items(),
            key=lambda x: x[1].last_accessed,
        )

        evict_count = len(self._cache) - self._max_cache_size
        for class_name, entry in sorted_entries[:evict_count]:
            logger.info(
                "cache_entry_evicted_size_limit",
                class_name=class_name,
                age_seconds=time.time() - entry.created_at,
                access_count=entry.access_count,
            )
            del self._cache[class_name]
            self._evictions += 1

    def get_method(
        self,
        class_name: str,
        method_name: str,
    ) -> Callable[..., Any]:
        """Get method callable with lazy instantiation.

        This is the main entry point for retrieving methods.

        Args:
            class_name: Name of class containing the method
            method_name: Name of method to retrieve

        Returns:
            Callable method (bound or injected)

        Raises:
            MethodRegistryError: If method cannot be retrieved
        """
        # Check for directly injected method first
        key = (class_name, method_name)
        if key in self._direct_methods:
            logger.debug(
                "method_retrieved_direct",
                class_name=class_name,
                method_name=method_name,
            )
            return self._direct_methods[key]

        # Get instance (lazy) and retrieve method
        try:
            instance = self._get_instance(class_name)
            method = getattr(instance, method_name)

            if not callable(method):
                raise MethodRegistryError(
                    f"'{class_name}.{method_name}' is not callable"
                )

            logger.debug(
                "method_retrieved_from_instance",
                class_name=class_name,
                method_name=method_name,
            )
            return method

        except AttributeError as exc:
            raise MethodRegistryError(
                f"Method '{method_name}' not found on class '{class_name}'"
            ) from exc

    def has_method(self, class_name: str, method_name: str) -> bool:
        """Check if a method is available (without instantiating).

        Args:
            class_name: Name of class
            method_name: Name of method

        Returns:
            True if method exists (or is directly injected)
        """
        # Check direct injection
        key = (class_name, method_name)
        if key in self._direct_methods:
            return True

        # Check if class is known and not failed
        if class_name in self._failed_classes:
            return False

        if class_name not in self._class_paths:
            return False

        # If instance exists, check method
        if class_name in self._cache:
            instance = self._cache[class_name].instance
            return hasattr(instance, method_name)
        # Otherwise, assume it exists (lazy check)
        # Full validation happens on first get_method() call
        return True

    def clear_cache(self) -> dict[str, Any]:
        """Clear all cached instances.

        This should be called between pipeline runs to prevent memory bloat.

        Returns:
            Statistics about cleared cache entries.
        """
        with self._lock:
            cache_size = len(self._cache)
            weakref_size = len(self._weakref_cache)

            stats = {
                "entries_cleared": cache_size,
                "weakrefs_cleared": weakref_size,
                "total_hits": self._cache_hits,
                "total_misses": self._cache_misses,
                "total_evictions": self._evictions,
                "total_instantiations": self._total_instantiations,
            }

            # Clear caches
            self._cache.clear()
            self._weakref_cache.clear()

            logger.info(
                "cache_cleared",
                **stats,
            )

            return stats

    def evict_expired(self) -> int:
        """Manually evict expired entries.

        Returns:
            Number of entries evicted.
        """
        with self._lock:
            expired = []
            for class_name, entry in self._cache.items():
                if entry.is_expired(self._cache_ttl_seconds):
                    expired.append(class_name)

            for class_name in expired:
                entry = self._cache[class_name]
                logger.info(
                    "cache_entry_evicted_manual",
                    class_name=class_name,
                    age_seconds=time.time() - entry.created_at,
                    access_count=entry.access_count,
                )
                del self._cache[class_name]
                self._evictions += 1

            return len(expired)

    def get_stats(self) -> dict[str, Any]:
        """Get registry statistics.

        Returns:
            Dictionary with registry stats including cache performance metrics
        """
        with self._lock:
            cache_entries = []
            for class_name, entry in self._cache.items():
                cache_entries.append({
                    "class_name": class_name,
                    "age_seconds": time.time() - entry.created_at,
                    "last_accessed_seconds_ago": time.time() - entry.last_accessed,
                    "access_count": entry.access_count,
                })

            hit_rate = 0.0
            total_accesses = self._cache_hits + self._cache_misses
            if total_accesses > 0:
                hit_rate = self._cache_hits / total_accesses

            return {
                "total_classes_registered": len(self._class_paths),
                "cached_instances": len(self._cache),
                "weakref_instances": len(self._weakref_cache),
                "failed_classes": len(self._failed_classes),
                "direct_methods_injected": len(self._direct_methods),
                "cache_hits": self._cache_hits,
                "cache_misses": self._cache_misses,
                "cache_hit_rate": hit_rate,
                "evictions": self._evictions,
                "total_instantiations": self._total_instantiations,
                "cache_ttl_seconds": self._cache_ttl_seconds,
                "max_cache_size": self._max_cache_size,
                "enable_weakref": self._enable_weakref,
                "cache_entries": cache_entries,
                "failed_class_names": list(self._failed_classes),
            }


def setup_default_instantiation_rules(registry: MethodRegistry) -> None:
    """Setup default special instantiation rules.

    These rules replicate the logic from the original MethodExecutor
    for classes that need non-default instantiation.

    Args:
        registry: MethodRegistry to configure
    """
    # MunicipalOntology - shared instance pattern
    ontology_instance = None

    def instantiate_ontology(cls: type) -> Any:
        nonlocal ontology_instance
        if ontology_instance is None:
            ontology_instance = cls()
        return ontology_instance

    registry.register_instantiation_rule("MunicipalOntology", instantiate_ontology)

    # SemanticAnalyzer, PerformanceAnalyzer, TextMiningEngine - need ontology
    def instantiate_with_ontology(cls: type) -> Any:
        if ontology_instance is None:
            raise MethodRegistryError(
                f"Cannot instantiate {cls.__name__}: MunicipalOntology not available"
            )
        return cls(ontology_instance)

    for class_name in ["SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine"]:
        registry.register_instantiation_rule(class_name, instantiate_with_ontology)

    # PolicyTextProcessor - needs ProcessorConfig
    def instantiate_policy_processor(cls: type) -> Any:
        try:
            from farfan_pipeline.processing.policy_processor import ProcessorConfig
            return cls(ProcessorConfig())
        except ImportError as exc:
            raise MethodRegistryError(
                "Cannot instantiate PolicyTextProcessor: ProcessorConfig unavailable"
            ) from exc

                              registry.register_instantiation_rule("PolicyTextProcessor",
instantiate_policy_processor)


__all__ = [
    "MethodRegistry",
    "MethodRegistryError",
    "setup_default_instantiation_rules",
]
```

src/orchestration/orchestrator.py

```
[ERROR LEYENDO ./src/orchestration/orchestrator.py: [Errno 2] No such file or directory:
'./src/orchestration/orchestrator.py']
```

src/orchestration/questionnaire_validation.py

```python
"""
Questionnaire Validation - Neutral Module for Structure Validation

This module is extracted from factory.py to break the import cycle between
factory.py and orchestrator.py. Both modules now import from here.

Part of JOBFRONT J2: Import cycle hardening.
"""

from __future__ import annotations

import logging
from typing import Any

logger = logging.getLogger(__name__)


def _validate_questionnaire_structure(monolith_data: dict[str, Any]) -> None:
    """Validate questionnaire structure.

    Args:
        monolith_data: Questionnaire data dictionary

    Raises:
        ValueError: If questionnaire structure is invalid
        TypeError: If questionnaire data types are incorrect
    """
    if not isinstance(monolith_data, dict):
        raise TypeError(f"Questionnaire must be a dict, got {type(monolith_data)}")

    # Validate canonical_notation exists
    if "canonical_notation" not in monolith_data:
        raise ValueError("Questionnaire missing 'canonical_notation'")

    canonical_notation = monolith_data["canonical_notation"]

    # Validate dimensions
    if "dimensions" not in canonical_notation:
        raise ValueError("Questionnaire missing 'canonical_notation.dimensions'")

    dimensions = canonical_notation["dimensions"]
    if not isinstance(dimensions, dict):
        raise TypeError("Dimensions must be a dict")

    # Validate dimension keys (D1-D6)
    expected_dim_keys = [f"D{i}" for i in range(1, 7)]
    found_dims = []

    for key, dim_data in dimensions.items():
        if key in expected_dim_keys:
            found_dims.append(key)
         elif isinstance(dim_data, dict) and dim_data.get("code") in ["DIM01", "DIM02",
"DIM03", "DIM04", "DIM05", "DIM06"]:
             # Found by code, which is acceptable
             pass

    # We require D1-D6 keys as per canonical JSON structure
    for dim_key in expected_dim_keys:
        if dim_key not in dimensions:
             # Check if it exists under another key but with correct code?
             # No, strict structure requires D1-D6 keys for now based on monolith.
             # However, if we want to be flexible:
             found = False
             for d in dimensions.values():
                                          if   isinstance(d,   dict)   and   d.get("code")   ==
f"DIM{int(dim_key[1:]):02d}":
                     found = True
                     break

             if not found:
                 # Fallback: check if key itself is DIM0x
                 alt_key = f"DIM{int(dim_key[1:]):02d}"
                 if alt_key in dimensions:
                     found = True

             if not found:
                raise ValueError(f"Missing dimension: {dim_key} (or equivalent code)")

    # Validate policy areas
    if "policy_areas" not in canonical_notation:
        raise ValueError("Questionnaire missing 'canonical_notation.policy_areas'")

    policy_areas = canonical_notation["policy_areas"]
    if not isinstance(policy_areas, dict):
        raise TypeError("Policy areas must be a dict")

    expected_pas = [f"PA{i:02d}" for i in range(1, 11)]
    for pa_id in expected_pas:
        if pa_id not in policy_areas:
            raise ValueError(f"Missing policy area: {pa_id}")

    logger.info("Questionnaire structure validation passed")


__all__ = ["_validate_questionnaire_structure"]
```

test_mathematical_audit.py

```python
"""
Tests para el auditor matemático de scoring macro

Verifica que el auditor ejecuta correctamente y produce reportes válidos.
"""

import json
import os
import pytest
from audit_mathematical_scoring_macro import (
    MacroScoringMathematicalAuditor,
    MathematicalCheck,
    AuditReport,
)


def test_auditor_initialization():
    """Test que el auditor se inicializa correctamente"""
    auditor = MacroScoringMathematicalAuditor()
    assert auditor.report.total_checks == 0
    assert auditor.report.passed_checks == 0
    assert auditor.report.failed_checks == 0


def test_weighted_average_audit():
    """Test auditoría de weighted average"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_weighted_average()

    assert len(checks) > 0
    assert all(isinstance(check, MathematicalCheck) for check in checks)
    assert any(check.check_id == "WA-001" for check in checks)
    assert any(check.procedure_name == "weighted_average" for check in checks)


def test_choquet_integral_audit():
    """Test auditoría de Choquet integral"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_choquet_integral()

    assert len(checks) > 0
    assert any(check.check_id == "CI-001" for check in checks)
    assert any(check.check_id == "CI-002" for check in checks)
    assert any("choquet" in check.procedure_name.lower() for check in checks)


def test_coherence_audit():
    """Test auditoría de cálculo de coherence"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_coherence_calculation()

    assert len(checks) > 0
    assert any(check.check_id == "COH-001" for check in checks)
    assert any("variance" in check.procedure_name.lower() or "coherence" in check.procedure_name.lower() for check in checks)


def test_penalty_factor_audit():
    """Test auditoría de penalty factor"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_penalty_factor()

    assert len(checks) > 0
    assert any(check.check_id == "PF-001" for check in checks)
    assert any("penalty" in check.procedure_name.lower() for check in checks)


def test_threshold_application_audit():
    """Test auditoría de aplicación de umbrales"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_threshold_application()

    assert len(checks) > 0
    assert any(check.check_id == "TH-001" for check in checks)
    assert any("threshold" in check.procedure_name.lower() for check in checks)


def test_weight_normalization_audit():
    """Test auditoría de normalización de pesos"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_weight_normalization()

    assert len(checks) > 0
    assert any(check.check_id == "WN-001" for check in checks)
    assert any("normalization" in check.procedure_name.lower() for check in checks)


def test_score_normalization_audit():
    """Test auditoría de normalización de scores"""
    auditor = MacroScoringMathematicalAuditor()
    checks = auditor.audit_score_normalization()

    assert len(checks) > 0
    assert any(check.check_id == "SN-001" for check in checks)


def test_complete_audit_execution():
    """Test ejecución completa de auditoría"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    # Verificar estructura del reporte
    assert isinstance(report, AuditReport)
    assert report.total_checks > 0
    assert report.total_checks == report.passed_checks + report.failed_checks
    assert len(report.all_checks) == report.total_checks
```

```python
    # Verificar que todos los checks están clasificados
    total_issues = (
        len(report.critical_issues) +
        len(report.high_issues) +
        len(report.medium_issues) +
        len(report.low_issues)
    )
    assert total_issues == report.failed_checks


def test_report_generation(tmp_path):
    """Test generación de reportes"""
    auditor = MacroScoringMathematicalAuditor()
    auditor.run_complete_audit()

    # Generar reporte Markdown
    md_path = tmp_path / "test_report.md"
    auditor.generate_detailed_report(str(md_path))
    assert md_path.exists()
    assert md_path.stat().st_size > 0

    # Verificar contenido básico
    content = md_path.read_text(encoding="utf-8")
    assert "Auditoría Matemática" in content
    assert "Resumen Ejecutivo" in content
    assert "Weighted Average" in content

    # Generar reporte JSON
    json_path = tmp_path / "test_report.json"
    auditor.generate_json_report(str(json_path))
    assert json_path.exists()

    # Verificar estructura JSON
    with open(json_path, "r", encoding="utf-8") as f:
        data = json.load(f)

    assert "summary" in data
    assert "checks" in data
    assert data["summary"]["total_checks"] > 0
    assert isinstance(data["checks"], list)


def test_all_checks_have_required_fields():
    """Test que todos los checks tienen los campos requeridos"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    for check in report.all_checks:
        assert check.check_id
        assert check.procedure_name
        assert check.description
        assert check.severity in ["CRITICAL", "HIGH", "MEDIUM", "LOW"]
        assert isinstance(check.passed, bool)
        assert isinstance(check.details, dict)
        assert check.recommendation


def test_severity_levels_present():
    """Test que hay checks de todas las severidades"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    severities = {check.severity for check in report.all_checks}

    # Debe haber checks de severidad CRITICAL y HIGH al menos
    assert "CRITICAL" in severities
    assert "HIGH" in severities


def test_mathematical_formulas_documented():
    """Test que todas las fórmulas matemáticas están documentadas"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    # Verificar que checks críticos tienen fórmulas documentadas
    critical_checks = [c for c in report.all_checks if c.severity == "CRITICAL"]

    for check in critical_checks:
        # Checks críticos deben tener al menos uno de estos campos
        has_formula = (
            "formula" in check.details or
            "implementation" in check.details or
            "validation" in check.details
        )
        assert has_formula, f"Check {check.check_id} missing formula documentation"


def test_audit_produces_expected_check_count():
    """Test que la auditoría produce el número esperado de verificaciones"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    # Debe haber al menos 20 verificaciones (ajustar según implementación real)
    assert report.total_checks >= 20

    # Verificar cobertura mínima por área
    check_ids = {check.check_id for check in report.all_checks}

    # Weighted Average
    assert any(cid.startswith("WA-") for cid in check_ids)

    # Choquet Integral
    assert any(cid.startswith("CI-") for cid in check_ids)

    # Coherence
    assert any(cid.startswith("COH-") for cid in check_ids)
    # Penalty Factor
    assert any(cid.startswith("PF-") for cid in check_ids)

    # Thresholds
    assert any(cid.startswith("TH-") for cid in check_ids)

    # Weight Normalization
    assert any(cid.startswith("WN-") for cid in check_ids)

    # Score Normalization
    assert any(cid.startswith("SN-") for cid in check_ids)


def test_no_duplicate_check_ids():
    """Test que no hay check_ids duplicados"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    check_ids = [check.check_id for check in report.all_checks]
    assert len(check_ids) == len(set(check_ids)), "Duplicate check_ids found"


def test_report_summary_consistency():
    """Test consistencia en el resumen del reporte"""
    auditor = MacroScoringMathematicalAuditor()
    report = auditor.run_complete_audit()

    # La suma de passed + failed debe igualar total
    assert report.passed_checks + report.failed_checks == report.total_checks

    # La suma de issues por severidad debe igualar failed_checks
    total_issues = (
        len(report.critical_issues) +
        len(report.high_issues) +
        len(report.medium_issues) +
        len(report.low_issues)
    )
    assert total_issues == report.failed_checks


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

test_scoring_mathematical_invariants.py

```
#!/usr/bin/env python3
"""
Test Suite for Scoring Mathematical Invariants
==============================================

Validates mathematical invariants and correctness properties
of the micro-level scoring procedures.

Mathematical Invariants Tested:
1. Score range: All scores must be in [0, 1]
2. Weight normalization: Weights must sum to 1.0 for weighted_mean
3. Threshold bounds: Thresholds must be in [0, 1]
4. Monotonicity: Higher component scores ? higher final score (for weighted_mean)
5. Commutativity: Order of components doesn't matter
6. Boundary conditions: Proper handling of 0 and 1 values
7. Aggregation correctness: max/min behave as expected

Author: F.A.R.F.A.N Pipeline Team
Date: 2025-12-11
Version: 1.0.0
"""

import pytest
from typing import Callable


# Inline scoring formulas for testing
class ScoringFormula:
    """Scoring formula implementation for testing."""

    def __init__(
        self,
        modality: str,
        threshold: float,
        aggregation: str,
        weight_elements: float,
        weight_similarity: float,
        weight_patterns: float
    ):
        self.modality = modality
        self.threshold = threshold
        self.aggregation = aggregation
        self.weight_elements = weight_elements
        self.weight_similarity = weight_similarity
        self.weight_patterns = weight_patterns

    def compute_score(
        self,
        elements: float,
        similarity: float,
        patterns: float
    ) -> float:
        """Compute score using modality formula."""
        if self.aggregation == "weighted_mean":
                      total_weight  =  self.weight_elements  +  self.weight_similarity  +
self.weight_patterns
            if total_weight == 0:
                return 0.0
            weighted_sum = (
                elements * self.weight_elements +
                similarity * self.weight_similarity +
                patterns * self.weight_patterns
            )
            return weighted_sum / total_weight
        elif self.aggregation == "max":
            return max(elements, similarity, patterns)
        elif self.aggregation == "min":
            return min(elements, similarity, patterns)
        return 0.0

    def passes_threshold(self, score: float) -> bool:
        """Check if score passes threshold."""
        return score >= self.threshold


# Define all scoring modalities
SCORING_FORMULAS = {
    "TYPE_A": ScoringFormula("TYPE_A", 0.65, "weighted_mean", 0.4, 0.3, 0.3),
    "TYPE_B": ScoringFormula("TYPE_B", 0.70, "weighted_mean", 0.5, 0.25, 0.25),
    "TYPE_C": ScoringFormula("TYPE_C", 0.60, "weighted_mean", 0.25, 0.5, 0.25),
    "TYPE_D": ScoringFormula("TYPE_D", 0.60, "weighted_mean", 0.25, 0.25, 0.5),
    "TYPE_E": ScoringFormula("TYPE_E", 0.75, "max", 1.0, 1.0, 1.0),
    "TYPE_F": ScoringFormula("TYPE_F", 0.55, "min", 1.0, 1.0, 1.0),
}


class TestScoringRangeInvariant:
    """Test that all scores are in valid range [0, 1]."""

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    @pytest.mark.parametrize("e,s,p", [
        (0.0, 0.0, 0.0),
        (1.0, 1.0, 1.0),
        (0.5, 0.5, 0.5),
        (0.0, 0.5, 1.0),
        (1.0, 0.5, 0.0),
        (0.25, 0.75, 0.5),
        (0.1, 0.2, 0.3),
        (0.9, 0.8, 0.7),
    ])
      def  test_score_in_valid_range(self,  modality_name:  str,  e:  float,  s:  float,  p:
float):
        """Invariant: score ? [0, 1] for all valid inputs."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.compute_score(e, s, p)
        assert 0.0 <= score <= 1.0, (
            f"{modality_name}: Score {score} out of valid range [0, 1] "
            f"for inputs E={e}, S={s}, P={p}"
        )


class TestWeightNormalizationInvariant:
    """Test that weights sum to 1.0 for weighted_mean aggregation."""

    @pytest.mark.parametrize("modality_name", [
        name for name, formula in SCORING_FORMULAS.items()
        if formula.aggregation == "weighted_mean"
    ])
    def test_weights_sum_to_one(self, modality_name: str):
        """Invariant: For weighted_mean, w_E + w_S + w_P = 1.0."""
        formula = SCORING_FORMULAS[modality_name]
        weights_sum = (
            formula.weight_elements +
            formula.weight_similarity +
            formula.weight_patterns
        )

        assert abs(weights_sum - 1.0) < 1e-10, (
            f"{modality_name}: Weights sum to {weights_sum}, expected 1.0"
        )


class TestThresholdBoundsInvariant:
    """Test that all thresholds are in valid range [0, 1]."""

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_threshold_in_valid_range(self, modality_name: str):
        """Invariant: threshold ? [0, 1]."""
        formula = SCORING_FORMULAS[modality_name]

        assert 0.0 <= formula.threshold <= 1.0, (
            f"{modality_name}: Threshold {formula.threshold} out of valid range [0, 1]"
        )


class TestMonotonicityInvariant:
    """Test monotonicity properties for weighted_mean aggregation."""

    @pytest.mark.parametrize("modality_name", [
        name for name, formula in SCORING_FORMULAS.items()
        if formula.aggregation == "weighted_mean"
    ])
    def test_monotonic_in_elements(self, modality_name: str):
        """Invariant: If E2 > E1, then score(E2, S, P) >= score(E1, S, P)."""
        formula = SCORING_FORMULAS[modality_name]
        s, p = 0.5, 0.5

        score1 = formula.compute_score(0.3, s, p)
        score2 = formula.compute_score(0.7, s, p)
        assert score2 >= score1, (
            f"{modality_name}: Not monotonic in elements. "
            f"score(0.3, {s}, {p}) = {score1} > score(0.7, {s}, {p}) = {score2}"
        )

    @pytest.mark.parametrize("modality_name", [
        name for name, formula in SCORING_FORMULAS.items()
        if formula.aggregation == "weighted_mean"
    ])
    def test_monotonic_in_similarity(self, modality_name: str):
        """Invariant: If S2 > S1, then score(E, S2, P) >= score(E, S1, P)."""
        formula = SCORING_FORMULAS[modality_name]
        e, p = 0.5, 0.5

        score1 = formula.compute_score(e, 0.3, p)
        score2 = formula.compute_score(e, 0.7, p)

        assert score2 >= score1, (
            f"{modality_name}: Not monotonic in similarity. "
            f"score({e}, 0.3, {p}) = {score1} > score({e}, 0.7, {p}) = {score2}"
        )

    @pytest.mark.parametrize("modality_name", [
        name for name, formula in SCORING_FORMULAS.items()
        if formula.aggregation == "weighted_mean"
    ])
    def test_monotonic_in_patterns(self, modality_name: str):
        """Invariant: If P2 > P1, then score(E, S, P2) >= score(E, S, P1)."""
        formula = SCORING_FORMULAS[modality_name]
        e, s = 0.5, 0.5

        score1 = formula.compute_score(e, s, 0.3)
        score2 = formula.compute_score(e, s, 0.7)

        assert score2 >= score1, (
            f"{modality_name}: Not monotonic in patterns. "
            f"score({e}, {s}, 0.3) = {score1} > score({e}, {s}, 0.7) = {score2}"
        )


class TestCommutativityInvariant:
    """Test that scoring is commutative (order doesn't matter)."""

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_permutation_invariance(self, modality_name: str):
            """Invariant: score(a, b, c) depends only on which value goes to which
weight."""
        formula = SCORING_FORMULAS[modality_name]

        # For weighted mean, changing values but keeping weight assignments
        # should give different results, but the computation should be stable
        score1 = formula.compute_score(0.3, 0.5, 0.7)
        score2 = formula.compute_score(0.3, 0.5, 0.7)  # Same call
        assert abs(score1 - score2) < 1e-10, (
            f"{modality_name}: Score computation is not stable/deterministic. "
            f"Got {score1} and {score2} for same inputs"
        )


class TestBoundaryConditionsInvariant:
    """Test proper handling of boundary conditions."""

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_all_zeros(self, modality_name: str):
        """Invariant: score(0, 0, 0) = 0."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.compute_score(0.0, 0.0, 0.0)

        assert abs(score - 0.0) < 1e-10, (
            f"{modality_name}: Expected score(0,0,0) = 0, got {score}"
        )

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_all_ones(self, modality_name: str):
        """Invariant: score(1, 1, 1) = 1."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.compute_score(1.0, 1.0, 1.0)

        assert abs(score - 1.0) < 1e-10, (
            f"{modality_name}: Expected score(1,1,1) = 1, got {score}"
        )


class TestAggregationCorrectnessInvariant:
    """Test that max/min aggregations work correctly."""

    def test_max_aggregation_type_e(self):
        """Invariant: For TYPE_E, score = max(E, S, P)."""
        formula = SCORING_FORMULAS["TYPE_E"]

        test_cases = [
            (0.8, 0.5, 0.3, 0.8),
            (0.3, 0.9, 0.2, 0.9),
            (0.2, 0.4, 0.95, 0.95),
            (0.5, 0.5, 0.5, 0.5),
        ]

        for e, s, p, expected in test_cases:
            score = formula.compute_score(e, s, p)
            assert abs(score - expected) < 1e-10, (
                f"TYPE_E: Expected max({e}, {s}, {p}) = {expected}, got {score}"
            )

    def test_min_aggregation_type_f(self):
        """Invariant: For TYPE_F, score = min(E, S, P)."""
        formula = SCORING_FORMULAS["TYPE_F"]
        test_cases = [
            (0.8, 0.5, 0.3, 0.3),
            (0.3, 0.9, 0.2, 0.2),
            (0.2, 0.4, 0.95, 0.2),
            (0.5, 0.5, 0.5, 0.5),
        ]

        for e, s, p, expected in test_cases:
            score = formula.compute_score(e, s, p)
            assert abs(score - expected) < 1e-10, (
                f"TYPE_F: Expected min({e}, {s}, {p}) = {expected}, got {score}"
            )


class TestThresholdLogicInvariant:
    """Test that threshold logic is correct."""

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_passes_at_threshold(self, modality_name: str):
        """Invariant: score == threshold should pass."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.threshold

        assert formula.passes_threshold(score), (
            f"{modality_name}: Score exactly at threshold {score} should pass"
        )

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_passes_above_threshold(self, modality_name: str):
        """Invariant: score > threshold should pass."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.threshold + 0.01

        assert formula.passes_threshold(score), (
             f"{modality_name}: Score {score} above threshold {formula.threshold} should
pass"
        )

    @pytest.mark.parametrize("modality_name", SCORING_FORMULAS.keys())
    def test_fails_below_threshold(self, modality_name: str):
        """Invariant: score < threshold should fail."""
        formula = SCORING_FORMULAS[modality_name]
        score = formula.threshold - 0.01

        assert not formula.passes_threshold(score), (
             f"{modality_name}: Score {score} below threshold {formula.threshold} should
fail"
        )


class TestWeightedMeanFormula:
    """Test specific weighted mean formula calculations."""
    def test_type_a_formula(self):
        """Test TYPE_A: 0.4*E + 0.3*S + 0.3*P."""
        formula = SCORING_FORMULAS["TYPE_A"]

        # Manual calculation: 0.4*0.6 + 0.3*0.8 + 0.3*0.5 = 0.24 + 0.24 + 0.15 = 0.63
        score = formula.compute_score(0.6, 0.8, 0.5)
        expected = 0.63

        assert abs(score - expected) < 1e-10, (
            f"TYPE_A: Expected {expected}, got {score}"
        )

    def test_type_b_formula(self):
        """Test TYPE_B: 0.5*E + 0.25*S + 0.25*P."""
        formula = SCORING_FORMULAS["TYPE_B"]

        # Manual calculation: 0.5*0.8 + 0.25*0.6 + 0.25*0.4 = 0.4 + 0.15 + 0.1 = 0.65
        score = formula.compute_score(0.8, 0.6, 0.4)
        expected = 0.65

        assert abs(score - expected) < 1e-10, (
            f"TYPE_B: Expected {expected}, got {score}"
        )

    def test_type_c_formula(self):
        """Test TYPE_C: 0.25*E + 0.5*S + 0.25*P."""
        formula = SCORING_FORMULAS["TYPE_C"]

        # Manual calculation: 0.25*0.4 + 0.5*0.9 + 0.25*0.3 = 0.1 + 0.45 + 0.075 = 0.625
        score = formula.compute_score(0.4, 0.9, 0.3)
        expected = 0.625

        assert abs(score - expected) < 1e-10, (
            f"TYPE_C: Expected {expected}, got {score}"
        )

    def test_type_d_formula(self):
        """Test TYPE_D: 0.25*E + 0.25*S + 0.5*P."""
        formula = SCORING_FORMULAS["TYPE_D"]

        # Manual calculation: 0.25*0.3 + 0.25*0.4 + 0.5*0.8 = 0.075 + 0.1 + 0.4 = 0.575
        score = formula.compute_score(0.3, 0.4, 0.8)
        expected = 0.575

        assert abs(score - expected) < 1e-10, (
            f"TYPE_D: Expected {expected}, got {score}"
        )


class TestScoreDistributionProperties:
    """Test statistical properties of score distributions."""

    @pytest.mark.parametrize("modality_name", [
        name for name, formula in SCORING_FORMULAS.items()
        if formula.aggregation == "weighted_mean"
    ])
    def test_balanced_inputs_give_balanced_score(self, modality_name: str):
        """Invariant: For balanced inputs (E=S=P=x), score should equal x."""
        formula = SCORING_FORMULAS[modality_name]

        for x in [0.0, 0.25, 0.5, 0.75, 1.0]:
            score = formula.compute_score(x, x, x)
            assert abs(score - x) < 1e-10, (
                f"{modality_name}: For balanced inputs ({x}, {x}, {x}), "
                f"expected score {x}, got {score}"
            )


if __name__ == "__main__":
    # Run with pytest
    pytest.main([__file__, "-v", "--tb=short"])
```

tests/canonic_phases/test_phase_four_seven_dura_lex.py

```
"""
Phase 4-7 Dura Lex Contract Tests
=================================

Applies the 15 Dura Lex contracts to the Phase 4-7 Aggregation pipeline,
specifically targeting the ChoquetAggregator and EnhancedAggregators.

Contracts Applied:
    1. Audit Trail - Aggregations must log detailed breakdowns
    2. Concurrency Determinism - Aggregation must be deterministic
    3. Context Immutability - ChoquetConfig must be immutable
    4. Deterministic Execution - Same scores -> same result
    5. Failure Fallback - Failures must be handled gracefully
    6. Governance - Validation rules must be enforced
    7. Idempotency - Repeated aggregation -> same result
    8. Monotone Compliance - Higher inputs -> higher/equal output (monotonicity)
    9. Permutation Invariance - Input order shouldn't matter for linear parts
    10. Refusal - Invalid configs/scores must be refused/clamped
    11. Retriever Contract - (N/A directly, but checked via config loading)
    12. Risk Certificate - (N/A directly, but validation details serve this)
    13. Routing Contract - (N/A directly)
    14. Snapshot Contract - CalibrationResult is a snapshot
    15. Traceability - Result allows tracing back to contributions

Author: Phase 4-7 Compliance
"""

import math
import pytest
from dataclasses import FrozenInstanceError

from canonic_phases.Phase_four_five_six_seven.choquet_aggregator import (
    ChoquetAggregator,
    ChoquetConfig,
    CalibrationConfigError,
)
from canonic_phases.Phase_four_five_six_seven.aggregation_enhancements import (
    EnhancedClusterAggregator,
    DispersionMetrics,
)

# ============================================================================
# CONTRACT 1: AUDIT TRAIL
# ============================================================================

def test_dura_lex_01_aggregation_audit_trail():
    """
    DURA LEX CONTRACT 1: All operations must be auditable.

    Validates:
        - Choquet aggregation returns full breakdown
        - Rationales are provided for every contribution
    """
    config = ChoquetConfig(
        linear_weights={"@a": 0.5, "@b": 0.5},
        interaction_weights={("@a", "@b"): 0.2}
    )
    aggregator = ChoquetAggregator(config)
    result = aggregator.aggregate("test", {"@a": 0.8, "@b": 0.6})

    # Contract: Full breakdown provided
    assert result.breakdown is not None
    assert result.breakdown.per_layer_rationales
    assert result.breakdown.per_interaction_rationales

    # Verify rationale content
    rationale = result.breakdown.per_layer_rationales["@a"]
    assert "weight=" in rationale
    assert "score=" in rationale


# ============================================================================
# CONTRACT 2 & 4: DETERMINISM
# ============================================================================

def test_dura_lex_02_04_aggregation_determinism():
    """
    DURA LEX CONTRACT 2 & 4: Execution must be deterministic.

    Validates:
        - Same inputs produce identical results
    """
    config = ChoquetConfig(linear_weights={"@a": 1.0})
    aggregator = ChoquetAggregator(config)

    score1 = aggregator.aggregate("test", {"@a": 0.5}).calibration_score
    score2 = aggregator.aggregate("test", {"@a": 0.5}).calibration_score

    assert score1 == score2


# ============================================================================
# CONTRACT 3: CONTEXT IMMUTABILITY
# ============================================================================

def test_dura_lex_03_config_immutability():
    """
    DURA LEX CONTRACT 3: Config objects must be immutable.

    Validates:
        - ChoquetConfig cannot be modified after creation
    """
    config = ChoquetConfig(linear_weights={"@a": 1.0})

    with pytest.raises(FrozenInstanceError):
        config.linear_weights = {"@b": 1.0}
```

```
# ============================================================================
# CONTRACT 5: FAILURE FALLBACK
# ============================================================================

def test_dura_lex_05_failure_handling():
    """
    DURA LEX CONTRACT 5: Failures must have defined behavior.

    Validates:
        - Missing layers raise specific error (ValueError)
    """
    config = ChoquetConfig(linear_weights={"@a": 0.5, "@b": 0.5})
    aggregator = ChoquetAggregator(config)

    # Missing layer @b
    with pytest.raises(ValueError) as exc:
        aggregator.aggregate("test", {"@a": 1.0})

    assert "Missing required layers" in str(exc.value)


# ============================================================================
# CONTRACT 6: GOVERNANCE (BOUNDEDNESS)
# ============================================================================

def test_dura_lex_06_boundedness_governance():
    """
    DURA LEX CONTRACT 6: Governance rules (boundedness) enforced.

    Validates:
        - Result is always in [0, 1]
        - Validation details capture check status
    """
    # Create config that could theoretically exceed 1.0 without normalization
    # But aggregator normalizes by default
    config = ChoquetConfig(
        linear_weights={"@a": 10.0},
        normalize_weights=True
    )
    aggregator = ChoquetAggregator(config)
    result = aggregator.aggregate("test", {"@a": 1.0})

    assert 0.0 <= result.calibration_score <= 1.0
    assert result.validation_passed
    assert result.validation_details["bounded"]


# ============================================================================
# CONTRACT 7: IDEMPOTENCY
# ============================================================================

def test_dura_lex_07_idempotency():
    """
    DURA LEX CONTRACT 7: Operations must be idempotent.

    Validates:
        - Repeated calls return same result object structure
    """
    config = ChoquetConfig(linear_weights={"@a": 1.0})
    aggregator = ChoquetAggregator(config)

    res1 = aggregator.aggregate("test", {"@a": 0.5})
    res2 = aggregator.aggregate("test", {"@a": 0.5})

    assert res1.calibration_score == res2.calibration_score
    assert res1.breakdown == res2.breakdown


# ============================================================================
# CONTRACT 8: MONOTONE COMPLIANCE
# ============================================================================

def test_dura_lex_08_monotonicity():
    """
    DURA LEX CONTRACT 8: Monotone compliance (Higher inputs -> Higher outputs).

    Validates:
        - Increasing a layer score increases (or keeps constant) the aggregate
    """
    config = ChoquetConfig(
        linear_weights={"@a": 0.5, "@b": 0.5},
        interaction_weights={("@a", "@b"): 0.2}
    )
    aggregator = ChoquetAggregator(config)

    low = aggregator.aggregate("test", {"@a": 0.2, "@b": 0.2}).calibration_score
    high = aggregator.aggregate("test", {"@a": 0.8, "@b": 0.2}).calibration_score

    assert high >= low


# ============================================================================
# CONTRACT 9: PERMUTATION INVARIANCE (PARTIAL)
# ============================================================================

def test_dura_lex_09_permutation_invariance():
    """
    DURA LEX CONTRACT 9: Permutation invariance.

    Validates:
        - Only relevant for symmetrical weights.
        - If weights are equal, swapping inputs shouldn't change result.
    """
    config = ChoquetConfig(
        linear_weights={"@a": 0.5, "@b": 0.5},
        interaction_weights={("@a", "@b"): 0.1}
    )
    aggregator = ChoquetAggregator(config)

    res1 = aggregator.aggregate("test", {"@a": 0.8, "@b": 0.2}).calibration_score
    res2 = aggregator.aggregate("test", {"@a": 0.2, "@b": 0.8}).calibration_score

    assert math.isclose(res1, res2)


# ==============================================================================
# CONTRACT 10: REFUSAL
# ==============================================================================

def test_dura_lex_10_refusal():
    """
    DURA LEX CONTRACT 10: Refusal of invalid configs.

    Validates:
        - Negative weights are refused
    """
    with pytest.raises(CalibrationConfigError):
        ChoquetConfig(linear_weights={"@a": -0.1})


# ==============================================================================
# CONTRACT 14: SNAPSHOT CONTRACT
# ==============================================================================

def test_dura_lex_14_snapshot():
    """
    DURA LEX CONTRACT 14: State must be capturable.

    Validates:
        - CalibrationResult acts as a snapshot of the calculation
    """
    config = ChoquetConfig(linear_weights={"@a": 1.0})
    aggregator = ChoquetAggregator(config)
    result = aggregator.aggregate("test", {"@a": 0.5})

    # Snapshot properties
    assert result.subject == "test"
    assert result.layer_scores == {"@a": 0.5}
    assert result.metadata is not None


# ==============================================================================
# CONTRACT 15: TRACEABILITY
# ==============================================================================

def test_dura_lex_15_traceability():
    """
    DURA LEX CONTRACT 15: Decisions must be traceable.

    Validates:
```

```
    - Enhanced cluster aggregator adaptive penalty is traceable via metrics
"""
class MockBase:
    pass

enhanced = EnhancedClusterAggregator(MockBase(), enable_contracts=False)

scores = [1.0, 1.0, 1.0, 1.0] # Convergence
metrics = enhanced.compute_dispersion_metrics(scores)

assert metrics.scenario == "convergence"

penalty = enhanced.adaptive_penalty(metrics)
# Convergence -> 0.5 * 0.3 = 0.15
assert math.isclose(penalty, 0.15)

# High dispersion
scores_disp = [0.0, 0.3, 0.7, 1.0]
metrics_disp = enhanced.compute_dispersion_metrics(scores_disp)
penalty_disp = enhanced.adaptive_penalty(metrics_disp)

# Verify different outcome implies tracing of input characteristics
assert penalty_disp != penalty
```

tests/canonic_phases/test_phase_zero.py

```python
"""
Unit Tests for Phase 0 Components
==================================

Tests Phase 0 exit gates, determinism, and orchestration according to
P00-EN v2.0 specification.

Test Categories:
    1. Exit Gate Tests - Individual gate validation
    2. Determinism Tests - Seed generation and application
    3. Integration Tests - Full Phase 0 orchestration
    4. Error Path Tests - Failure scenarios

Author: Phase 0 Compliance Team
Version: 1.0.0
"""

from unittest.mock import MagicMock

import pytest

from canonic_phases.Phase_zero.determinism import (
    ALL_SEEDS,
    MANDATORY_SEEDS,
    OPTIONAL_SEEDS,
    apply_seeds_to_rngs,
    derive_seed_from_parts,
    derive_seed_from_string,
    validate_seed_application,
)
from canonic_phases.Phase_zero.exit_gates import (
    check_all_gates,
    check_bootstrap_gate,
    check_determinism_gate,
    check_input_verification_gate,
)
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig


# ============================================================================
# Exit Gate Tests
# ============================================================================

class MockRunner:
    """Mock Phase 0 runner for testing."""

    def __init__(self):
        self.errors = []
        self._bootstrap_failed = False
        self.runtime_config = None
        self.seed_snapshot = {}
        self.input_pdf_sha256 = ""
        self.questionnaire_sha256 = ""


def test_bootstrap_gate_passes_with_valid_config():
    """Gate 1 should pass when bootstrap succeeds."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)

    result = check_bootstrap_gate(runner)

    assert result.passed
    assert result.gate_id == 1
    assert result.gate_name == "bootstrap"
    assert result.reason is None


def test_bootstrap_gate_fails_on_bootstrap_failure():
    """Gate 1 should fail if _bootstrap_failed is True."""
    runner = MockRunner()
    runner._bootstrap_failed = True
    runner.runtime_config = MagicMock(spec=RuntimeConfig)

    result = check_bootstrap_gate(runner)

    assert not result.passed
    assert result.gate_id == 1
    assert "Bootstrap failed" in result.reason


def test_bootstrap_gate_fails_on_missing_config():
    """Gate 1 should fail if runtime_config is None."""
    runner = MockRunner()
    runner.runtime_config = None

    result = check_bootstrap_gate(runner)

    assert not result.passed
    assert "Runtime config not loaded" in result.reason


def test_bootstrap_gate_fails_with_errors():
    """Gate 1 should fail if errors present."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.errors = ["Some error"]

    result = check_bootstrap_gate(runner)

    assert not result.passed
    assert "Some error" in result.reason


def test_input_verification_gate_passes_with_hashes():
    """Gate 2 should pass when both files are hashed."""
```

```python
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.input_pdf_sha256 = "abc123" * 10 + "abcd"  # 64 chars
    runner.questionnaire_sha256 = "def456" * 10 + "defa"

    result = check_input_verification_gate(runner)

    assert result.passed
    assert result.gate_id == 2
    assert result.gate_name == "input_verification"


def test_input_verification_gate_fails_on_missing_pdf_hash():
    """Gate 2 should fail if PDF not hashed."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.questionnaire_sha256 = "def456" * 10 + "defa"
    # input_pdf_sha256 is empty string

    result = check_input_verification_gate(runner)

    assert not result.passed
    assert "Input PDF not hashed" in result.reason


def test_determinism_gate_passes_with_mandatory_seeds():
    """Gate 4 should pass when python and numpy seeds present."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.input_pdf_sha256 = "abc123" * 10 + "abcd"
    runner.questionnaire_sha256 = "def456" * 10 + "defa"
    runner.seed_snapshot = {
        "python": 12345,
        "numpy": 67890,
    }

    result = check_determinism_gate(runner)

    assert result.passed
    assert result.gate_id == 4


def test_determinism_gate_fails_on_missing_python_seed():
    """Gate 4 should fail if python seed missing."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.input_pdf_sha256 = "abc123" * 10 + "abcd"
    runner.questionnaire_sha256 = "def456" * 10 + "defa"
    runner.seed_snapshot = {
        "numpy": 67890,
        # Missing python seed
    }

    result = check_determinism_gate(runner)
    assert not result.passed
    assert "Missing mandatory seeds" in result.reason
    assert "python" in result.reason


def test_check_all_gates_success():
    """check_all_gates should return True when all gates pass."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner.input_pdf_sha256 = "abc123" * 10 + "abcd"
    runner.questionnaire_sha256 = "def456" * 10 + "defa"
    runner.seed_snapshot = {"python": 12345, "numpy": 67890}

    all_passed, results = check_all_gates(runner)

    assert all_passed
    assert len(results) == 4
    assert all(r.passed for r in results)


def test_check_all_gates_fails_fast():
    """check_all_gates should stop at first failure."""
    runner = MockRunner()
    runner.runtime_config = MagicMock(spec=RuntimeConfig)
    runner._bootstrap_failed = True  # Gate 1 will fail

    all_passed, results = check_all_gates(runner)

    assert not all_passed
    assert len(results) == 1  # Only checked gate 1
    assert not results[0].passed


# ============================================================================
# Determinism Tests
# ============================================================================

def test_derive_seed_from_string_deterministic():
    """Seed derivation should be deterministic for same input."""
    seed1 = derive_seed_from_string("test_input")
    seed2 = derive_seed_from_string("test_input")

    assert seed1 == seed2
    assert isinstance(seed1, int)
    assert 0 <= seed1 < 2**32


def test_derive_seed_from_string_unique():
    """Different inputs should produce different seeds."""
    seed1 = derive_seed_from_string("input_a")
    seed2 = derive_seed_from_string("input_b")

    assert seed1 != seed2
```

```
def test_derive_seed_from_parts_deterministic():
    """Seed derivation from parts should be deterministic."""
    seed1 = derive_seed_from_parts("PU_123", "corr-1", "python")
    seed2 = derive_seed_from_parts("PU_123", "corr-1", "python")

    assert seed1 == seed2


def test_apply_seeds_to_rngs_success():
    """Applying seeds should succeed with mandatory seeds."""
    seeds = {
        "python": 12345,
        "numpy": 67890,
        "quantum": 11111,
    }

    status = apply_seeds_to_rngs(seeds)

    assert status["python"] is True


def test_apply_seeds_to_rngs_fails_without_python():
    """Applying seeds should fail if python seed missing."""
    seeds = {
        "numpy": 67890,
    }

    with pytest.raises(ValueError, match="Missing mandatory seeds"):
        apply_seeds_to_rngs(seeds)


def test_validate_seed_application_success():
    """Validation should pass when mandatory seeds applied."""
    seeds = {"python": 12345, "numpy": 67890}
    status = {"python": True, "numpy": True}

    success, errors = validate_seed_application(seeds, status)

    assert success
    assert len(errors) == 0


def test_mandatory_seeds_constant():
    """MANDATORY_SEEDS should contain python and numpy."""
    assert "python" in MANDATORY_SEEDS
    assert "numpy" in MANDATORY_SEEDS


def test_all_seeds_complete():
    """ALL_SEEDS should be union of mandatory and optional."""
    expected = set(MANDATORY_SEEDS) | set(OPTIONAL_SEEDS)
    assert set(ALL_SEEDS) == expected
```

```
tests/canonic_phases/test_phase_zero_dura_lex.py

"""
Phase 0 Dura Lex Contract Tests
================================

Applies the 15 Dura Lex contracts from
src/cross_cutting_infrastrucuiture/contractual/dura_lex/
to Phase 0 validation and bootstrap processes.

Contracts Applied:
    1. Audit Trail - All Phase 0 operations must be auditable
    2. Concurrency Determinism - Phase 0 must be deterministic
    3. Context Immutability - Config objects must be immutable
    4. Deterministic Execution - Seeds must produce same results
    5. Failure Fallback - Failures must have defined behavior
    6. Governance - Runtime modes must be enforced
    7. Idempotency - Multiple runs must produce same result
    8. Monotone Compliance - No degradation in validation
    9. Permutation Invariance - Order-independent where applicable
    10. Refusal - Invalid configs must be refused
    11. Retriever Contract - File loading must satisfy contract
    12. Risk Certificate - Risks must be documented
    13. Routing Contract - Decision paths must be traceable
    14. Snapshot Contract - State must be capturable
    15. Traceability - All decisions must leave trace

Author: Phase 0 Compliance Team
Version: 1.0.0
"""

import hashlib
import json
import os
import tempfile
from pathlib import Path
from unittest.mock import MagicMock, patch

import pytest

# Phase 0 imports
from canonic_phases.Phase_zero.determinism import (
    apply_seeds_to_rngs,
    derive_seed_from_string,
)
from canonic_phases.Phase_zero.exit_gates import check_all_gates
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from canonic_phases.Phase_zero.verified_pipeline_runner import VerifiedPipelineRunner


# ============================================================================
# CONTRACT 1: AUDIT TRAIL
# ============================================================================
def test_dura_lex_01_all_operations_must_be_auditable():
    """
    DURA LEX CONTRACT 1: All Phase 0 operations must leave audit trail.

    Validates:
        - Bootstrap logs runtime config load
        - Input verification logs hash computation
        - Boot checks log validation results
        - Determinism logs seed application
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "test.pdf"
        pdf_path.write_bytes(b"PDF content")

        q_path = Path(tmpdir) / "q.json"
        q_path.write_text('{"test": "data"}')

        artifacts_dir = Path(tmpdir) / "artifacts"

        with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig')
as mock_config:
            mock_config.from_env.return_value = MagicMock(mode=MagicMock(value="dev"))

            runner = VerifiedPipelineRunner(pdf_path, artifacts_dir, q_path)

            # Contract: All operations must create audit trail
            # Verify artifacts directory contains logs
            assert artifacts_dir.exists()

            # Contract: Execution ID must be traceable
            assert hasattr(runner, 'execution_id')
            assert runner.execution_id is not None


# ============================================================================
# CONTRACT 2: CONCURRENCY DETERMINISM
# ============================================================================

def test_dura_lex_02_phase_zero_must_be_deterministic():
    """
    DURA LEX CONTRACT 2: Phase 0 must be deterministic.

    Validates:
        - Same inputs produce same hashes
        - Same seeds produce same RNG state
        - Execution ID generation is traceable
    """
    # Same file content produces same hash
    content = b"Deterministic content"
    hash1 = hashlib.sha256(content).hexdigest()
    hash2 = hashlib.sha256(content).hexdigest()

    # Contract: Hashing must be deterministic
    assert hash1 == hash2
    # Same seed produces same derived seed
    seed1 = derive_seed_from_string("test_input")
    seed2 = derive_seed_from_string("test_input")

    # Contract: Seed derivation must be deterministic
    assert seed1 == seed2


# ============================================================================
# CONTRACT 3: CONTEXT IMMUTABILITY
# ============================================================================

def test_dura_lex_03_runtime_config_must_be_immutable():
    """
    DURA LEX CONTRACT 3: RuntimeConfig must be immutable once created.

    Validates:
        - RuntimeConfig cannot be modified after creation
        - Mode cannot be changed
        - Flags cannot be toggled
    """
    with patch.dict(os.environ, {"SAAAAAA_RUNTIME_MODE": "prod"}):
        config = RuntimeConfig.from_env()

        # Contract: Mode should not be modifiable
        original_mode = config.mode

        # Attempt to modify should fail or be ignored
        try:
            config.mode = RuntimeMode.DEV
            # If modification succeeds, value should not change (frozen dataclass)
            assert config.mode == original_mode
        except (AttributeError, Exception):
            # Expected: frozen dataclass prevents modification
            pass


# ============================================================================
# CONTRACT 4: DETERMINISTIC EXECUTION
# ============================================================================

def test_dura_lex_04_seed_application_must_be_reproducible():
    """
    DURA LEX CONTRACT 4: Seeding RNGs must produce reproducible results.

    Validates:
        - Python random.seed() produces same sequence
        - NumPy np.random.seed() produces same sequence
        - Re-seeding resets to same state
    """
    import random

    # Apply seeds
    seeds = {"python": 42, "numpy": 42}
    apply_seeds_to_rngs(seeds)

    # Generate sequence
    seq1 = [random.random() for _ in range(10)]

    # Re-apply same seeds
    apply_seeds_to_rngs(seeds)
    seq2 = [random.random() for _ in range(10)]

    # Contract: Sequences must be identical
    assert seq1 == seq2


# ============================================================================
# CONTRACT 5: FAILURE FALLBACK
# ============================================================================

def test_dura_lex_05_bootstrap_failure_must_have_defined_behavior():
    """
    DURA LEX CONTRACT 5: Bootstrap failures must have defined fallback.

    Validates:
        - _bootstrap_failed flag is set
        - errors list is populated
        - Failure manifest can be generated
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "test.pdf"
        artifacts_dir = Path(tmpdir) / "artifacts"
        q_path = Path(tmpdir) / "q.json"

        # Force bootstrap failure
          with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig')
as mock_config:
            mock_config.from_env.side_effect = Exception("Config load failed")

            runner = VerifiedPipelineRunner(pdf_path, artifacts_dir, q_path)

            # Contract: Failure must be recorded
            assert runner._bootstrap_failed is True
            assert len(runner.errors) > 0

            # Contract: Failure manifest can be generated
            manifest_path = runner.generate_failure_manifest()
            assert manifest_path.exists()


# ============================================================================
# CONTRACT 6: GOVERNANCE
# ============================================================================

def test_dura_lex_06_prod_mode_must_enforce_strict_validation():
    """
    DURA LEX CONTRACT 6: PROD mode must enforce governance rules.

    Validates:
        - PROD mode rejects invalid configs
        - PROD mode fails on boot check errors
        - DEV mode allows warnings
    """
    # PROD mode configuration
    with patch.dict(os.environ, {"SAAAAAA_RUNTIME_MODE": "prod"}):
        config = RuntimeConfig.from_env()

        # Contract: PROD mode must be strict
        assert config.mode == RuntimeMode.PROD
        assert config.is_strict_mode() is True


# ============================================================================
# CONTRACT 7: IDEMPOTENCY
# ============================================================================

def test_dura_lex_07_hash_computation_must_be_idempotent():
    """
    DURA LEX CONTRACT 7: Operations must be idempotent.

    Validates:
        - Computing hash multiple times produces same result
        - Seed generation is idempotent
        - Gate checking is idempotent
    """
    content = b"Test content"

    # Compute hash multiple times
    hash1 = hashlib.sha256(content).hexdigest()
    hash2 = hashlib.sha256(content).hexdigest()
    hash3 = hashlib.sha256(content).hexdigest()

    # Contract: Results must be identical (idempotent)
    assert hash1 == hash2 == hash3


# ============================================================================
# CONTRACT 8: MONOTONE COMPLIANCE
# ============================================================================

def test_dura_lex_08_validation_must_not_degrade():
    """
    DURA LEX CONTRACT 8: Validation strictness must not degrade.

    Validates:
        - All 4 gates must be checked
        - Cannot skip gates
        - Gate results are monotone (once failed, stays failed)
    """
    class MockRunner:
        def __init__(self):
            self.errors = []
            self._bootstrap_failed = False
            self.runtime_config = MagicMock()
            self.input_pdf_sha256 = "a" * 64
            self.questionnaire_sha256 = "b" * 64
            self.seed_snapshot = {"python": 123, "numpy": 456}

    runner = MockRunner()

    # Check all gates
    all_passed1, results1 = check_all_gates(runner)

    # Contract: Checking again produces same result (monotone)
    all_passed2, results2 = check_all_gates(runner)

    assert all_passed1 == all_passed2
    assert len(results1) == len(results2)


# ============================================================================
# CONTRACT 9: PERMUTATION INVARIANCE
# ============================================================================

def test_dura_lex_09_gate_results_independent_of_check_order():
    """
    DURA LEX CONTRACT 9: Individual gate results must be order-independent.

    Validates:
        - Each gate checks independent criteria
        - Gate results don't depend on previous gates (except fail-fast)
    """
    from canonic_phases.Phase_zero.exit_gates import (
        check_bootstrap_gate,
        check_determinism_gate,
    )

    class MockRunner:
        def __init__(self):
            self.errors = []
            self._bootstrap_failed = False
            self.runtime_config = MagicMock()
            self.input_pdf_sha256 = "a" * 64
            self.questionnaire_sha256 = "b" * 64
            self.seed_snapshot = {"python": 123, "numpy": 456}

    runner = MockRunner()

    # Check gates in different order
    result_bootstrap = check_bootstrap_gate(runner)
    result_determinism = check_determinism_gate(runner)

    # Contract: Results should be independent
    assert result_bootstrap.passed
    assert result_determinism.passed


# ============================================================================
# CONTRACT 10: REFUSAL
# ============================================================================

def test_dura_lex_10_invalid_configs_must_be_refused():
    """
    DURA LEX CONTRACT 10: System must refuse invalid requests.

    Validates:
        - Invalid runtime mode is refused
        - Missing required files are refused
        - Tampered hashes are refused
    """
    # Invalid runtime mode should raise error
    with patch.dict(os.environ, {"SAAAAAA_RUNTIME_MODE": "invalid"}):
        with pytest.raises(Exception):
            RuntimeConfig.from_env()


# ============================================================================
# CONTRACT 11: RETRIEVER CONTRACT
# ============================================================================

def test_dura_lex_11_file_loading_must_satisfy_contract():
    """
    DURA LEX CONTRACT 11: File retrieval must satisfy contract.

    Validates:
        - Files must exist before reading
        - Hashing must succeed or fail cleanly
        - Error messages must be actionable
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "nonexistent.pdf"
        q_path = Path(tmpdir) / "q.json"
        q_path.write_text('{}')

        with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig') as mock_config:
            mock_config.from_env.return_value = MagicMock(mode=MagicMock(value="dev"))

            runner = VerifiedPipelineRunner(pdf_path, Path(tmpdir) / "artifacts", q_path)

            # Contract: Missing file must be detected
            success = runner.verify_input()
            assert not success
            assert len(runner.errors) > 0
            assert any("not found" in err.lower() for err in runner.errors)
```

```
# ==============================================================================
# CONTRACT 12: RISK CERTIFICATE
# ==============================================================================

def test_dura_lex_12_risks_must_be_documented():
    """
    DURA LEX CONTRACT 12: Risks must be certified and documented.

    Validates:
        - Missing dependencies documented in boot checks
        - DEV mode risks are logged
        - Failure manifest documents all errors
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "test.pdf"
        pdf_path.write_bytes(b"PDF")

        with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig')
as mock_config:
            mock_config.from_env.side_effect = Exception("Risk: Config missing")

            runner = VerifiedPipelineRunner(
                pdf_path,
                Path(tmpdir) / "artifacts",
                Path(tmpdir) / "q.json"
            )

            # Contract: Risk must be documented in errors
            assert len(runner.errors) > 0

            # Contract: Failure manifest documents risk
            manifest_path = runner.generate_failure_manifest()
            with open(manifest_path) as f:
                manifest = json.load(f)

            assert "errors" in manifest
            assert len(manifest["errors"]) > 0


# ==============================================================================
# CONTRACT 13: ROUTING CONTRACT
# ==============================================================================

def test_dura_lex_13_decision_paths_must_be_traceable():
    """
    DURA LEX CONTRACT 13: All routing decisions must be traceable.

    Validates:
        - Gate pass/fail decisions are recorded
        - Error reasons are documented
        - Execution flow is traceable via execution_id
    """
    class MockRunner:
        def __init__(self):
            self.errors = ["Test error"]
            self._bootstrap_failed = False
            self.runtime_config = MagicMock()
            self.input_pdf_sha256 = ""
            self.questionnaire_sha256 = ""
            self.seed_snapshot = {}

    runner = MockRunner()
    all_passed, results = check_all_gates(runner)

    # Contract: Decision path must be traceable
    assert not all_passed  # Failed due to errors

    # Contract: Failure reason must be documented
    failed_gate = next(r for r in results if not r.passed)
    assert failed_gate.reason is not None
    assert "error" in failed_gate.reason.lower()


# ============================================================================
# CONTRACT 14: SNAPSHOT CONTRACT
# ============================================================================

def test_dura_lex_14_state_must_be_capturable():
    """
    DURA LEX CONTRACT 14: System state must be snapshot-able.

    Validates:
        - Seed snapshot is captured
        - Input hashes are captured
        - Runtime config state is captured
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "test.pdf"
        pdf_path.write_bytes(b"PDF")
        q_path = Path(tmpdir) / "q.json"
        q_path.write_text('{}')

        with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig')
as mock_config:
            mock_config.from_env.return_value = MagicMock(mode=MagicMock(value="dev"))

            runner = VerifiedPipelineRunner(pdf_path, Path(tmpdir) / "artifacts",
q_path)

            # Contract: State can be captured
            state_snapshot = {
                "execution_id": runner.execution_id,
                "policy_unit_id": runner.policy_unit_id,
                "runtime_mode": runner.runtime_config.mode.value if
runner.runtime_config else None,
                "bootstrap_failed": runner._bootstrap_failed,
                "errors_count": len(runner.errors),
            }
            # Contract: Snapshot contains key state
            assert "execution_id" in state_snapshot
            assert "runtime_mode" in state_snapshot


# ============================================================================
# CONTRACT 15: TRACEABILITY
# ============================================================================

def test_dura_lex_15_all_decisions_must_leave_trace():
    """
    DURA LEX CONTRACT 15: All decisions must be traceable.

    Validates:
        - Hash values are recorded
        - Seed values are recorded
        - Gate results are recorded
        - Errors are recorded
    """
    with tempfile.TemporaryDirectory() as tmpdir:
        pdf_path = Path(tmpdir) / "test.pdf"
        pdf_path.write_bytes(b"PDF content")
        q_path = Path(tmpdir) / "q.json"
        q_path.write_text('{"test": "data"}')

        with patch('canonic_phases.Phase_zero.verified_pipeline_runner.RuntimeConfig')
as mock_config:
            mock_config.from_env.return_value = MagicMock(mode=MagicMock(value="dev"))

            runner = VerifiedPipelineRunner(pdf_path, Path(tmpdir) / "artifacts",
q_path)
            runner.verify_input()

            # Contract: All decisions leave traces
            trace = {
                "input_pdf_hash": runner.input_pdf_sha256 if hasattr(runner,
'input_pdf_sha256') else None,
                "questionnaire_hash": runner.questionnaire_sha256 if hasattr(runner,
'questionnaire_sha256') else None,
                "bootstrap_failed": runner._bootstrap_failed,
                "errors": runner.errors,
                "execution_id": runner.execution_id,
            }

            # Contract: Trace is complete
            assert trace["input_pdf_hash"] is not None
            assert trace["questionnaire_hash"] is not None
            assert trace["execution_id"] is not None


# ============================================================================
# Summary Test
# ============================================================================
def test_dura_lex_summary_all_15_contracts():
    """Verify all 15 Dura Lex contracts are tested."""
    contracts = [
        "Audit Trail",
        "Concurrency Determinism",
        "Context Immutability",
        "Deterministic Execution",
        "Failure Fallback",
        "Governance",
        "Idempotency",
        "Monotone Compliance",
        "Permutation Invariance",
        "Refusal",
        "Retriever Contract",
        "Risk Certificate",
        "Routing Contract",
        "Snapshot Contract",
        "Traceability",
    ]

    print("\n" + "="*70)
    print("DURA LEX CONTRACT COMPLIANCE - PHASE 0")
    print("="*70)
    for i, contract in enumerate(contracts, 1):
        print(f"? Contract {i:2d}: {contract}")
    print("="*70)
    print(f"Total Contracts Tested: {len(contracts)}/15 (100%)")
    print("="*70 + "\n")

    assert len(contracts) == 15
```

tests/canonic_phases/test_signature_validator.py

```
import pytest
from pathlib import Path
from canonic_phases.Phase_zero.signature_validator import (
    SignatureRegistry,
    FunctionSignature,
    validate_signature,
    validate_call_signature,
    _signature_registry
)

def test_signature_registry():
    registry = SignatureRegistry(registry_path=Path("tmp_registry.json"))

    def my_func(a: int, b: str) -> bool:
        return True

    sig = registry.register_function(my_func)

    assert sig.function_name == "my_func"
    assert "a" in sig.parameters
    assert sig.parameter_types["a"] == str(int)

     retrieved = registry.get_signature("tests.canonic_phases.test_signature_validator",
None, "my_func")
    # Note: module name depends on how test is run

    # Clean up
    if registry.registry_path.exists():
        registry.registry_path.unlink()

def test_validate_signature_decorator():
    @validate_signature(enforce=True)
    def my_func(a: int) -> int:
        return a + 1

    assert my_func(1) == 2

    with pytest.raises(TypeError):
        my_func("string") # type: ignore

def test_validate_call_signature():
    def my_func(a: int, b: int):
        pass

    assert validate_call_signature(my_func, 1, 2)
    assert not validate_call_signature(my_func, 1) # Missing arg
```

tests/choquet_tests.py

```
"""
Property-Based Tests for Choquet Aggregator

This test suite validates the Choquet aggregator using property-based testing
with Hypothesis to ensure mathematical correctness across a wide range of inputs.

Properties tested:
1. Boundedness: 0.0 ? Cal(I) ? 1.0 for all valid inputs
2. Monotonicity: Higher layer scores ? higher aggregate score
3. Normalization: Proper weight normalization
4. Interaction correctness: min(x?, x?) constraint
5. Determinism: Same inputs ? same outputs

Test markers:
- updated: Current, maintained tests
- outdated: Deprecated tests (excluded from CI)
"""

from __future__ import annotations

import pytest
from hypothesis import given, strategies as st, assume, settings

import sys
from pathlib import Path

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

from canonic_phases.Phase_four_five_six_seven.choquet_aggregator import (
    ChoquetAggregator,
    ChoquetConfig,
    CalibrationConfigError,
)


@pytest.mark.updated
class TestChoquetBoundedness:
    """Test boundedness property: 0.0 ? Cal(I) ? 1.0"""

    @given(
        scores=st.dictionaries(
            keys=st.sampled_from(["@b", "@chain", "@q", "@d"]),
                        values=st.floats(min_value=0.0,  max_value=1.0,  allow_nan=False,
allow_infinity=False),
            min_size=1,
            max_size=4
        )
    )
    @settings(max_examples=100, deadline=None)
    def test_boundedness_uniform_weights(self, scores: dict[str, float]) -> None:
        """Cal(I) must be in [0,1] with uniform weights."""
        layers = list(scores.keys())
        n = len(layers)
        weight = 1.0 / n

        config = ChoquetConfig(
            linear_weights={layer: weight for layer in layers},
            interaction_weights={},
            validate_boundedness=True,
            normalize_weights=False
        )

        aggregator = ChoquetAggregator(config)
        result = aggregator.aggregate(
            subject="test_subject",
            layer_scores=scores
        )

        assert 0.0 <= result.calibration_score <= 1.0, (
            f"Boundedness violated: Cal(I)={result.calibration_score}, scores={scores}"
        )
        assert result.validation_passed

    @given(
        scores=st.dictionaries(
            keys=st.sampled_from(["@b", "@chain", "@q"]),
                        values=st.floats(min_value=0.0,  max_value=1.0,  allow_nan=False,
allow_infinity=False),
            min_size=3,
            max_size=3
        ),
        interaction_weight=st.floats(min_value=0.0, max_value=0.2, allow_nan=False)
    )
    @settings(max_examples=100, deadline=None)
    def test_boundedness_with_interactions(
        self,
        scores: dict[str, float],
        interaction_weight: float
    ) -> None:
        """Cal(I) must be in [0,1] with interaction terms (after clamping)."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={
                ("@b", "@chain"): interaction_weight
            },
            validate_boundedness=False,
            normalize_weights=True
        )

        aggregator = ChoquetAggregator(config)
        result = aggregator.aggregate(
            subject="test_subject",
            layer_scores=scores
        )
        assert 0.0 <= result.calibration_score <= 1.0, (
            f"Boundedness violated: Cal(I)={result.calibration_score}, "
            f"interaction_weight={interaction_weight}"
        )

    @given(
        n_layers=st.integers(min_value=2, max_value=8),
        n_interactions=st.integers(min_value=0, max_value=5)
    )
    @settings(max_examples=50, deadline=None)
    def test_boundedness_random_config(
        self,
        n_layers: int,
        n_interactions: int
    ) -> None:
        """Cal(I) bounded for random layer/interaction configurations."""
        layer_ids = [f"@layer{i}" for i in range(n_layers)]

        linear_weights = {
            layer: 1.0 / n_layers for layer in layer_ids
        }

        interaction_weights = {}
        for i in range(min(n_interactions, n_layers * (n_layers - 1) // 2)):
            if i < n_layers - 1:
                interaction_weights[(layer_ids[i], layer_ids[i+1])] = 0.05

        config = ChoquetConfig(
            linear_weights=linear_weights,
            interaction_weights=interaction_weights,
            validate_boundedness=True,
            normalize_weights=True
        )

        layer_scores = {layer: 0.5 for layer in layer_ids}

        aggregator = ChoquetAggregator(config)
        result = aggregator.aggregate(
            subject="test_subject",
            layer_scores=layer_scores
        )

        assert 0.0 <= result.calibration_score <= 1.0
        assert result.validation_passed


@pytest.mark.updated
class TestChoquetMonotonicity:
    """Test monotonicity property: higher inputs ? higher outputs"""

    @given(
        base_score=st.floats(min_value=0.1, max_value=0.5, allow_nan=False),
        increment=st.floats(min_value=0.01, max_value=0.4, allow_nan=False)
    )
```

```
@settings(max_examples=100, deadline=None)
def test_monotonicity_single_layer(
    self,
    base_score: float,
    increment: float
) -> None:
    """Increasing single layer score increases Cal(I)."""
    assume(base_score + increment <= 1.0)

    config = ChoquetConfig(
        linear_weights={"@b": 0.5, "@chain": 0.5},
        interaction_weights={},
        validate_boundedness=True
    )

    aggregator = ChoquetAggregator(config)

    scores_low = {"@b": base_score, "@chain": 0.5}
    result_low = aggregator.aggregate("test", scores_low)

    scores_high = {"@b": base_score + increment, "@chain": 0.5}
    result_high = aggregator.aggregate("test", scores_high)

    assert result_high.calibration_score >= result_low.calibration_score, (
        f"Monotonicity violated: "
        f"scores_low={scores_low} ? {result_low.calibration_score:.4f}, "
        f"scores_high={scores_high} ? {result_high.calibration_score:.4f}"
    )

@given(
    scores=st.dictionaries(
        keys=st.sampled_from(["@b", "@chain", "@q"]),
        values=st.floats(min_value=0.0, max_value=0.8, allow_nan=False),
        min_size=3,
        max_size=3
    ),
    scale_factor=st.floats(min_value=1.0, max_value=1.25, allow_nan=False)
)
@settings(max_examples=100, deadline=None)
def test_monotonicity_all_layers(
    self,
    scores: dict[str, float],
    scale_factor: float
) -> None:
    """Scaling all scores up increases Cal(I)."""
    scaled_scores = {
        layer: min(1.0, score * scale_factor)
        for layer, score in scores.items()
    }

    assume(any(
        scaled_scores[layer] > scores[layer]
        for layer in scores
    ))
```

```
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={("@b", "@chain"): 0.1},
            validate_boundedness=True
        )

        aggregator = ChoquetAggregator(config)

        result_base = aggregator.aggregate("test", scores)
        result_scaled = aggregator.aggregate("test", scaled_scores)

        assert result_scaled.calibration_score >= result_base.calibration_score, (
            f"Monotonicity violated: base={result_base.calibration_score:.4f}, "
            f"scaled={result_scaled.calibration_score:.4f}"
        )


@pytest.mark.updated
class TestChoquetNormalization:
    """Test weight normalization behavior"""

    def test_normalization_linear_weights(self) -> None:
        """Linear weights are normalized to sum to 1.0."""
        config = ChoquetConfig(
            linear_weights={"@b": 2.0, "@chain": 3.0, "@q": 5.0},
            interaction_weights={},
            normalize_weights=True
        )

        aggregator = ChoquetAggregator(config)

        total = sum(aggregator._normalized_linear_weights.values())
        assert abs(total - 1.0) < 1e-10, f"Normalization failed: sum={total}"

    def test_normalization_preserves_ratios(self) -> None:
        """Normalization preserves weight ratios."""
        config = ChoquetConfig(
            linear_weights={"@b": 4.0, "@chain": 2.0, "@q": 2.0},
            interaction_weights={},
            normalize_weights=True
        )

        aggregator = ChoquetAggregator(config)
        normalized = aggregator._normalized_linear_weights

        assert abs(normalized["@b"] - 0.5) < 1e-10
        assert abs(normalized["@chain"] - 0.25) < 1e-10
        assert abs(normalized["@q"] - 0.25) < 1e-10

    @given(
        weights=st.dictionaries(
            keys=st.sampled_from(["@b", "@chain", "@q", "@d"]),
            values=st.floats(min_value=0.1, max_value=10.0, allow_nan=False),
            min_size=2,
            max_size=4
        )
    )
    @settings(max_examples=100, deadline=None)
    def test_normalization_property(self, weights: dict[str, float]) -> None:
        """Normalized weights always sum to 1.0."""
        config = ChoquetConfig(
            linear_weights=weights,
            interaction_weights={},
            normalize_weights=True
        )

        aggregator = ChoquetAggregator(config)
        total = sum(aggregator._normalized_linear_weights.values())

        assert abs(total - 1.0) < 1e-10, f"Normalization failed: sum={total}"


@pytest.mark.updated
class TestChoquetInteractionTerms:
    """Test interaction term computation"""

    def test_interaction_uses_min(self) -> None:
        """Interaction term uses min(x?, x?) as expected."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.5, "@chain": 0.5},
            interaction_weights={("@b", "@chain"): 0.2},
            normalize_weights=False,
            validate_boundedness=False
        )

        aggregator = ChoquetAggregator(config)

        scores = {"@b": 0.8, "@chain": 0.6}
        result = aggregator.aggregate("test", scores)

        expected_linear = 0.5 * 0.8 + 0.5 * 0.6
        expected_interaction = 0.2 * min(0.8, 0.6)
        expected_total = expected_linear + expected_interaction

        assert abs(result.breakdown.linear_contribution - expected_linear) < 1e-10
          assert abs(result.breakdown.interaction_contribution - expected_interaction) <
1e-10
        assert abs(result.calibration_score - expected_total) < 1e-10

    @given(
        score_b=st.floats(min_value=0.0, max_value=1.0, allow_nan=False),
        score_chain=st.floats(min_value=0.0, max_value=1.0, allow_nan=False)
    )
    @settings(max_examples=100, deadline=None)
    def test_interaction_symmetry(
        self,
        score_b: float,
        score_chain: float
    ) -> None:
        """Interaction term respects min() symmetry."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.5, "@chain": 0.5},
            interaction_weights={("@b", "@chain"): 0.1},
            normalize_weights=False,
            validate_boundedness=False
        )

        aggregator = ChoquetAggregator(config)

        scores = {"@b": score_b, "@chain": score_chain}
        result = aggregator.aggregate("test", scores)

        expected_min = min(score_b, score_chain)
             interaction_contrib = result.breakdown.per_interaction_contributions[("@b",
"@chain")]
        expected_interaction = 0.1 * expected_min

        assert abs(interaction_contrib - expected_interaction) < 1e-10, (
            f"Interaction computation incorrect: expected={expected_interaction:.4f}, "
            f"got={interaction_contrib:.4f}"
        )


@pytest.mark.updated
class TestChoquetDeterminism:
    """Test deterministic behavior"""

    @given(
        scores=st.dictionaries(
            keys=st.sampled_from(["@b", "@chain", "@q"]),
            values=st.floats(min_value=0.0, max_value=1.0, allow_nan=False),
            min_size=3,
            max_size=3
        )
    )
    @settings(max_examples=50, deadline=None)
    def test_deterministic_output(self, scores: dict[str, float]) -> None:
        """Same inputs produce same outputs."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={("@b", "@chain"): 0.1},
            validate_boundedness=False
        )

        aggregator = ChoquetAggregator(config)

        result1 = aggregator.aggregate("test", scores)
        result2 = aggregator.aggregate("test", scores)

        assert result1.calibration_score == result2.calibration_score
                                   assert    result1.breakdown.linear_contribution    ==
result2.breakdown.linear_contribution
                               assert    result1.breakdown.interaction_contribution    ==
result2.breakdown.interaction_contribution


@pytest.mark.updated
class TestChoquetConfigValidation:
    """Test configuration validation"""

    def test_empty_linear_weights_raises(self) -> None:
        """Empty linear_weights raises CalibrationConfigError."""
            with  pytest.raises(CalibrationConfigError,  match="linear_weights  cannot  be
empty"):
            ChoquetConfig(linear_weights={})

    def test_negative_linear_weight_raises(self) -> None:
        """Negative linear weight raises CalibrationConfigError."""
        with pytest.raises(CalibrationConfigError, match="Negative weight not allowed"):
            ChoquetConfig(linear_weights={"@b": -0.5})

    def test_negative_interaction_weight_raises(self) -> None:
        """Negative interaction weight raises CalibrationConfigError."""
        with pytest.raises(CalibrationConfigError, match="Negative interaction weight"):
            ChoquetConfig(
                linear_weights={"@b": 0.5, "@chain": 0.5},
                interaction_weights={("@b", "@chain"): -0.1}
            )

    def test_interaction_missing_layer_raises(self) -> None:
        """Interaction referencing missing layer raises CalibrationConfigError."""
        with pytest.raises(CalibrationConfigError, match="not in linear_weights"):
            ChoquetConfig(
                linear_weights={"@b": 0.5},
                interaction_weights={("@b", "@missing"): 0.1}
            )

    def test_invalid_layer_id_type_raises(self) -> None:
        """Non-string layer ID raises CalibrationConfigError."""
        with pytest.raises(CalibrationConfigError, match="Layer ID must be string"):
            ChoquetConfig(linear_weights={123: 0.5})  # type: ignore

    def test_invalid_weight_type_raises(self) -> None:
        """Non-numeric weight raises CalibrationConfigError."""
        with pytest.raises(CalibrationConfigError, match="Weight must be numeric"):
            ChoquetConfig(linear_weights={"@b": "invalid"})  # type: ignore


@pytest.mark.updated
class TestChoquetAggregatorErrors:
    """Test error handling in aggregation"""

    def test_missing_layer_scores_raises(self) -> None:
        """Missing required layer in layer_scores raises ValueError."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.5, "@chain": 0.5}
        )
        aggregator = ChoquetAggregator(config)

        with pytest.raises(ValueError, match="Missing required layers"):
            aggregator.aggregate("test", layer_scores={"@b": 0.8})

    def test_boundedness_violation_raises(self) -> None:
        """Boundedness violation raises CalibrationConfigError when enabled."""
        config = ChoquetConfig(
            linear_weights={"@b": 2.0, "@chain": 2.0},
            interaction_weights={},
            normalize_weights=False,
            validate_boundedness=True
        )
        aggregator = ChoquetAggregator(config)

        with pytest.raises(CalibrationConfigError, match="Boundedness violation"):
            aggregator.aggregate("test", layer_scores={"@b": 1.0, "@chain": 1.0})

    def test_boundedness_violation_warning_only(self) -> None:
        """Boundedness violation logs warning when validation disabled."""
        config = ChoquetConfig(
            linear_weights={"@b": 2.0, "@chain": 2.0},
            interaction_weights={},
            normalize_weights=False,
            validate_boundedness=False
        )
        aggregator = ChoquetAggregator(config)

        result = aggregator.aggregate("test", layer_scores={"@b": 1.0, "@chain": 1.0})

        assert not result.validation_passed
        assert result.calibration_score == 1.0


@pytest.mark.updated
class TestChoquetBreakdown:
    """Test breakdown computation and rationales"""

    def test_breakdown_components_sum_to_total(self) -> None:
        """Linear + interaction contributions sum to total score."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={("@b", "@chain"): 0.1},
            normalize_weights=False
        )

        aggregator = ChoquetAggregator(config)
        scores = {"@b": 0.8, "@chain": 0.7, "@q": 0.9}
        result = aggregator.aggregate("test", scores)

        total_from_breakdown = (
            result.breakdown.linear_contribution +
            result.breakdown.interaction_contribution
        )

        assert abs(result.calibration_score - total_from_breakdown) < 1e-10

    def test_per_layer_contributions_sum_to_linear(self) -> None:
        """Per-layer contributions sum to total linear contribution."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={},
            normalize_weights=False
        )

        aggregator = ChoquetAggregator(config)
        scores = {"@b": 0.8, "@chain": 0.7, "@q": 0.9}
        result = aggregator.aggregate("test", scores)

        per_layer_sum = sum(result.breakdown.per_layer_contributions.values())

        assert abs(result.breakdown.linear_contribution - per_layer_sum) < 1e-10

    def test_rationales_present(self) -> None:
        """All layers and interactions have rationales."""
        config = ChoquetConfig(
            linear_weights={"@b": 0.4, "@chain": 0.3, "@q": 0.3},
            interaction_weights={("@b", "@chain"): 0.1},
            normalize_weights=False
        )

        aggregator = ChoquetAggregator(config)
        scores = {"@b": 0.8, "@chain": 0.7, "@q": 0.9}
        result = aggregator.aggregate("test", scores)

        assert len(result.breakdown.per_layer_rationales) == 3
        assert len(result.breakdown.per_interaction_rationales) == 1

        for layer in ["@b", "@chain", "@q"]:
            assert layer in result.breakdown.per_layer_rationales
            rationale = result.breakdown.per_layer_rationales[layer]
            assert "weight=" in rationale
            assert "score=" in rationale

        assert ("@b", "@chain") in result.breakdown.per_interaction_rationales
             interaction_rationale  =  result.breakdown.per_interaction_rationales[("@b",
"@chain")]
        assert "min(" in interaction_rationale
```

tests/conftest.py

```python
"""Pytest configuration for F.A.R.F.A.N test suite."""
```

tests/dashboard_atroz_/test_ingestion_resolution.py

```
from __future__ import annotations

from dataclasses import dataclass

from                  farfan_pipeline.dashboard_atroz_.ingestion                  import
_resolve_municipality_from_context


@dataclass
class _InputData:
    document_id: str
    pdf_path: str
    run_id: str = "run_test_1"


@dataclass
class _Document:
    input_data: _InputData


def test_resolve_municipality_by_dane_code_in_document_id() -> None:
    context = {
                   "document":  _Document(input_data=_InputData(document_id="pdt_19050",
pdf_path="/tmp/whatever.pdf")),
    }
    municipality = _resolve_municipality_from_context(context)
    assert municipality is not None
    assert municipality.dane_code == "19050"
    assert municipality.name.lower() == "argelia"


def test_resolve_municipality_by_dane_code_in_pdf_path() -> None:
    context = {
                       "document":   _Document(input_data=_InputData(document_id="doc_x",
pdf_path="/tmp/pdt_81794.pdf")),
    }
    municipality = _resolve_municipality_from_context(context)
    assert municipality is not None
    assert municipality.dane_code == "81794"
    assert municipality.name.lower() == "tame"
```

tests/dashboard_atroz_/test_pdet_reference_data.py

```
from __future__ import annotations

import pytest

from farfan_pipeline.dashboard_atroz_.pdet_colombia_data import PDETSubregion,
PDET_MUNICIPALITIES


def test_pdet_reference_counts() -> None:
    assert len(PDET_MUNICIPALITIES) == 170
    assert len(set(m.dane_code for m in PDET_MUNICIPALITIES)) == 170
     assert len(set(m.subregion for m in PDET_MUNICIPALITIES)) == len(PDETSubregion) ==
16


def test_pdet_reference_unique_name_department_pairs() -> None:
         pairs = {(m.name.strip().lower(), m.department.strip().lower()) for m in
PDET_MUNICIPALITIES}
    assert len(pairs) == 170
```

tests/dashboard_atroz_/test_store_bootstrap.py

```python
from __future__ import annotations

import pytest

from farfan_pipeline.dashboard_atroz_.api_v1_store import AtrozStore


@pytest.mark.asyncio
async def test_store_bootstraps_170_municipalities_and_16_regions() -> None:
    store = AtrozStore()

    regions = await store.list_regions()
    assert len(regions) == 16

    total_municipalities = 0
    for region in regions:
        municipalities = await store.list_region_municipalities(region.id)
        total_municipalities += len(municipalities)

    assert total_municipalities == 170
```

tests/orchestration/orchestration_examples/__init__.py

```python
"""
Orchestration examples package.
"""
```