```
#!/usr/bin/env python3
"""
HYDRA SENTINEL-X: ENHANCED COGNITIVE AGENTS v2.0
True Multi-Agent Architecture with Dual-Process Thinking
Authority: President & C.E.O. Daryell McFarland
"""

import json
import math
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple, Any
from dataclasses import dataclass, field
from enum import Enum
from pathlib import Path
import numpy as np


class ThoughtProcess(Enum):
    """Dual-process thinking modes"""
    SYSTEM_1 = "FAST"  # Intuitive, pattern-based
    SYSTEM_2 = "SLOW"  # Deliberate, analytical


@dataclass
class CognitiveState:
    """Agent's mental state during decision-making"""
    initial_reaction: str
    retrieved_memories: List[Dict]
    memory_influence: float
    final_decision: str
    confidence: float
    reasoning_chain: List[str]
    emotional_state: Dict[str, float]
    override_occurred: bool = False


@dataclass
class AgentPersonality:
    """Personality trait configuration"""
    aggression: float  # 0.0 to 1.0 - Risk tolerance
    patience: float    # 0.0 to 1.0 - Wait for confirmation
    greed: float       # 0.0 to 1.0 - Profit seeking
    fear: float        # 0.0 to 1.0 - Loss aversion
    contrarian: float  # 0.0 to 1.0 - Tendency to go against crowd
```

```
    def to_dict(self) -> Dict[str, float]:
        return {
            'aggression': self.aggression,
            'patience': self.patience,
            'greed': self.greed,
            'fear': self.fear,
            'contrarian': self.contrarian
        }


class CognitiveAgent:
    """Enhanced agent with dual-process thinking"""

    def __init__(self, name: str, role: str, personality: AgentPersonality,
                 color_code: str = ""):
        self.name = name
        self.role = role
        self.personality = personality
        self.color_code = color_code
        self.decision_history: List[CognitiveState] = []
        self.learning_rate = 0.1

    def system_1_reaction(self, market_context: Dict[str, Any]) -> Tuple[str, flo
        """
        FAST thinking - Immediate pattern recognition based on personality
        Returns: (decision, confidence, reasoning)
        """
        price_change = market_context.get('price_change_pct', 0)
        rsi = market_context.get('rsi', 50)
        volume_spike = market_context.get('volume_spike', False)
        trend = market_context.get('trend', 'NEUTRAL')

        # Personality-driven heuristics
        decision_score = 0.0
        reasons = []

        # Aggression influences breakout response
        if volume_spike and abs(price_change) > 2:
            aggression_boost = self.personality.aggression * 2.0
            decision_score += aggression_boost if price_change > 0 else -aggressi
            reasons.append(f"Aggression ({self.personality.aggression:.1f}) → {'B

        # Patience requires more confirmation
        if self.personality.patience > 0.7:
            if trend == 'NEUTRAL':
                decision_score *= 0.5  # Reduce conviction in unclear trends
```

```
            reasons.append(f"Patience ({self.personality.patience:.1f}) → Wai

        # Greed chases momentum
        if self.personality.greed > 0.6 and price_change > 3:
            decision_score += self.personality.greed * 1.5
            reasons.append(f"Greed ({self.personality.greed:.1f}) → Chase momentu

        # Fear avoids losses
        if self.personality.fear > 0.6 and price_change < -2:
            decision_score -= self.personality.fear * 2.0
            reasons.append(f"Fear ({self.personality.fear:.1f}) → Avoid falling k

        # RSI-based reactions modulated by personality
        if rsi > 70:
            overbought_reaction = -1.0 * (1.0 - self.personality.contrarian)
            decision_score += overbought_reaction
            reasons.append(f"RSI {rsi:.1f} overbought → {'Fade' if not self.perso
        elif rsi < 30:
            oversold_reaction = 1.0 * (1.0 - self.personality.contrarian)
            decision_score += oversold_reaction
            reasons.append(f"RSI {rsi:.1f} oversold → {'Buy dip' if not self.pers

        # Convert score to decision
        if decision_score > 1.5:
            decision = "BUY"
            confidence = min(abs(decision_score) * 0.25, 0.95)
        elif decision_score < -1.5:
            decision = "SELL"
            confidence = min(abs(decision_score) * 0.25, 0.95)
        elif abs(decision_score) > 0.5:
            decision = "WAIT"
            confidence = 0.4
        else:
            decision = "HOLD"
            confidence = 0.3

        reasoning = " | ".join(reasons) if reasons else "Neutral market, no stron

        return decision, confidence, reasoning

    def system_2_deliberation(self, initial_decision: str, initial_confidence: fl
                              memory_system, market_context: Dict[str, Any]) -> T
        """
        SLOW thinking - Memory retrieval and deliberate reasoning
        Returns: (final_decision, final_confidence, memories, override_flag)
        """
        symbol = market_context.get('symbol', 'UNKNOWN')
```

```
# Retrieve relevant memories
context_tags = [
    symbol,
    initial_decision,
    market_context.get('trend', 'NEUTRAL'),
    "BREAKOUT" if market_context.get('volume_spike') else "NORMAL"
]

memories = memory_system.recall(context_tags, min_relevance=0.3)

if not memories:
    return initial_decision, initial_confidence, [], False

# Analyze memory sentiment
memory_sentiment_sum = 0.0
trauma_count = 0
win_count = 0

for mem_data in memories:
    mem = mem_data['memory']
    sentiment = mem.get('sentiment', 0.0)
    relevance = mem_data.get('relevance', 0.0)

    # Weight sentiment by relevance
    weighted_sentiment = sentiment * relevance
    memory_sentiment_sum += weighted_sentiment

    if sentiment < -0.5:
        trauma_count += 1
    elif sentiment > 0.5:
        win_count += 1

avg_memory_sentiment = memory_sentiment_sum / len(memories)

# Decision override logic
override = False
final_decision = initial_decision
final_confidence = initial_confidence

# Strong negative memories can override BUY signals
if initial_decision == "BUY" and trauma_count >= 2 and avg_memory_sentime
    override = True
    final_decision = "WAIT"
    final_confidence *= 0.5

# Strong positive memories can boost confidence
```

```
        elif initial_decision == "BUY" and win_count >= 2 and avg_memory_sentimen
            final_confidence = min(final_confidence * 1.3, 0.95)

        # Fear personality amplifies trauma response
        if trauma_count > 0 and self.personality.fear > 0.6:
            final_confidence *= (1.0 - self.personality.fear * 0.3)

        # Contrarian agents might flip on strong consensus
        if self.personality.contrarian > 0.7:
            if abs(avg_memory_sentiment) > 0.6:
                # Sometimes go against strong memory signals
                if np.random.random() < self.personality.contrarian * 0.3:
                    override = True
                    if final_decision == "BUY":
                        final_decision = "WAIT"
                    final_confidence *= 0.7

        return final_decision, final_confidence, memories, override

    def think(self, market_context: Dict[str, Any], memory_system) -> CognitiveSt
        """
        Complete thinking process: System 1 → System 2
        """
        # System 1: Fast reaction
        s1_decision, s1_confidence, s1_reasoning = self.system_1_reaction(market_

        reasoning_chain = [
            f"[SYSTEM 1] Initial reaction: {s1_decision} ({s1_confidence:.0%} con
            f"[SYSTEM 1] Reasoning: {s1_reasoning}"
        ]

        # System 2: Deliberate analysis
        s2_decision, s2_confidence, memories, override = self.system_2_deliberati
            s1_decision, s1_confidence, memory_system, market_context
        )

        if memories:
            reasoning_chain.append(f"[SYSTEM 2] Retrieved {len(memories)} relevan
            for i, mem_data in enumerate(memories[:2], 1):  # Top 2
                mem = mem_data['memory']
                reasoning_chain.append(
                    f"  Memory {i}: {mem['txt'][:60]}... (sentiment: {mem['sentim
                )

        if override:
            reasoning_chain.append(
                f"[SYSTEM 2] OVERRIDE: {s1_decision} → {s2_decision} (confidence:
```

```
            )
        else:
            reasoning_chain.append(
                f"[SYSTEM 2] Confirmed: {s2_decision} (confidence: {s2_confidence
            )

        # Calculate emotional state
        emotional_state = {
            'fear_level': self.personality.fear * (1.0 if s2_decision in ["WAIT",
            'greed_level': self.personality.greed * (1.0 if s2_decision == "BUY"
            'confidence_level': s2_confidence,
            'stress_level': 1.0 - self.personality.patience if override else 0.3
        }

        state = CognitiveState(
            initial_reaction=s1_decision,
            retrieved_memories=memories,
            memory_influence=abs(s2_confidence - s1_confidence),
            final_decision=s2_decision,
            confidence=s2_confidence,
            reasoning_chain=reasoning_chain,
            emotional_state=emotional_state,
            override_occurred=override
        )

        self.decision_history.append(state)
        return state

    def learn_from_outcome(self, decision_id: str, actual_pnl_pct: float):
        """
        Update personality based on outcomes (simple reinforcement learning)
        """
        # Find the decision in history
        # Adjust personality traits slightly based on success/failure

        if actual_pnl_pct > 5:  # Good profit
            # Reinforce the traits that led to success
            if self.personality.aggression > 0.5:
                self.personality.aggression = min(0.95, self.personality.aggressi
        elif actual_pnl_pct < -3:  # Loss
            # Increase fear, reduce aggression
            self.personality.fear = min(0.95, self.personality.fear + self.learni
            self.personality.aggression = max(0.05, self.personality.aggression -


class AgentCouncil:
    """Manages multi-agent decision making"""
```

```
def __init__(self, memory_system):
    self.memory = memory_system
    self.agents: List[CognitiveAgent] = []
    self._initialize_agents()

def _initialize_agents(self):
    """Create the four core agents"""

    # STINKMEANER: The Aggressor
    stinkmeaner = CognitiveAgent(
        name="STINKMEANER",
        role="The Aggressor",
        personality=AgentPersonality(
            aggression=0.9,
            patience=0.2,
            greed=0.8,
            fear=0.2,
            contrarian=0.3
        ),
        color_code="RED"
    )

    # SAMUEL: The Strategist
    samuel = CognitiveAgent(
        name="SAMUEL",
        role="The Strategist",
        personality=AgentPersonality(
            aggression=0.5,
            patience=0.8,
            greed=0.5,
            fear=0.4,
            contrarian=0.4
        ),
        color_code="BLUE"
    )

    # CLAYTON: The Conservator
    clayton = CognitiveAgent(
        name="CLAYTON",
        role="The Conservator",
        personality=AgentPersonality(
            aggression=0.2,
            patience=0.9,
            greed=0.3,
            fear=0.7,
            contrarian=0.2
```

```
            ),
            color_code="GREEN"
        )

        # JULIUS: The Trader
        julius = CognitiveAgent(
            name="JULIUS",
            role="The Trader",
            personality=AgentPersonality(
                aggression=0.6,
                patience=0.7,
                greed=0.6,
                fear=0.5,
                contrarian=0.5
            ),
            color_code="YELLOW"
        )

        self.agents = [stinkmeaner, samuel, clayton, julius]

    def deliberate(self, market_context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Full council deliberation process
        Returns consensus decision with metadata
        """
        agent_states = {}

        # Each agent independently analyzes
        for agent in self.agents:
            state = agent.think(market_context, self.memory)
            agent_states[agent.name] = state

        # Aggregate decisions
        decision_votes = {"BUY": 0, "SELL": 0, "HOLD": 0, "WAIT": 0}
        total_confidence = 0.0

        for agent_name, state in agent_states.items():
            decision = state.final_decision
            confidence = state.confidence

            # Weight vote by confidence
            decision_votes[decision] += confidence
            total_confidence += confidence

        # Determine consensus
        consensus_decision = max(decision_votes, key=decision_votes.get)
        consensus_strength = decision_votes[consensus_decision] / total_confidenc
```

```
        # Calculate average confidence
        avg_confidence = total_confidence / len(self.agents)

        # Count overrides
        override_count = sum(1 for state in agent_states.values() if state.overri

        # Unanimous decision bonus
        unique_decisions = set(state.final_decision for state in agent_states.val
        is_unanimous = len(unique_decisions) == 1

        if is_unanimous:
            final_confidence = min(avg_confidence * 1.2, 0.95)
        else:
            final_confidence = avg_confidence * consensus_strength

        return {
            'consensus_decision': consensus_decision,
            'consensus_strength': consensus_strength,
            'final_confidence': final_confidence,
            'agent_states': agent_states,
            'decision_votes': decision_votes,
            'override_count': override_count,
            'is_unanimous': is_unanimous,
            'avg_agent_confidence': avg_confidence
        }

    def explain_decision(self, deliberation_result: Dict[str, Any]) -> str:
        """Generate human-readable explanation"""
        lines = []
        lines.append("=" * 70)
        lines.append("AGENT COUNCIL DELIBERATION")
        lines.append("=" * 70)

        for agent_name, state in deliberation_result['agent_states'].items():
            agent = next(a for a in self.agents if a.name == agent_name)
            lines.append(f"\n{agent.name} ({agent.role}):")
            lines.append(f"  Decision: {state.final_decision} (Confidence: {state
            if state.override_occurred:
                lines.append(f"  ⚠️  OVERRIDE: {state.initial_reaction} → {state.
            lines.append(f"  Emotional State: Fear={state.emotional_state['fear_l
            for reason in state.reasoning_chain[:3]:
                lines.append(f"    • {reason}")

        lines.append("\n" + "=" * 70)
        lines.append("COUNCIL CONSENSUS")
        lines.append("=" * 70)
```

```
        lines.append(f"Decision: {deliberation_result['consensus_decision']}")
        lines.append(f"Strength: {deliberation_result['consensus_strength']:.0%}"
        lines.append(f"Final Confidence: {deliberation_result['final_confidence']
        lines.append(f"Unanimous: {'Yes' if deliberation_result['is_unanimous'] e
        lines.append(f"Overrides: {deliberation_result['override_count']}/4 agent

        lines.append("\nVote Distribution:")
        for decision, weight in deliberation_result['decision_votes'].items():
            lines.append(f"  {decision}: {weight:.2f}")

        return "\n".join(lines)


# Example usage demonstration
if __name__ == "__main__":
    from decimal import Decimal

    # Mock memory system for testing
    class MockMemory:
        def __init__(self):
            self.memories = []

        def recall(self, tags, min_relevance=0.3):
            # Return some fake memories
            return [
                {
                    'relevance': 0.8,
                    'memory': {
                        'txt': 'BTC fake breakout crashed -8%, stopped out',
                        'sentiment': -0.9,
                        'tags': tags
                    }
                },
                {
                    'relevance': 0.6,
                    'memory': {
                        'txt': 'Similar volume spike led to +12% gain',
                        'sentiment': 0.7,
                        'tags': tags
                    }
                }
            ]

    # Create council
    memory = MockMemory()
    council = AgentCouncil(memory)
```

```
# Test market scenario
market_context = {
    'symbol': 'BTC/USD',
    'price': 50000,
    'price_change_pct': 3.5,
    'rsi': 75,
    'volume_spike': True,
    'trend': 'BULLISH'
}

# Get decision
result = council.deliberate(market_context)
explanation = council.explain_decision(result)

print(explanation)
print(f"\n✅ EXECUTE: {result['consensus_decision']} with {result['final_conf
```