

ДОДАТОК А

Вихідний код класу PersonalitySnapshot (поєднання трьох шарів моделі)

Клас PersonalitySnapshot інкапсулює стан особистості однієї імперії та поєднує всі три шари моделі – статичний архетип, динамічний настрій і персистентну пам'ять – в один коригувальний множник, що повертається зміненому ШІ гри на час спрацювання хука. Метод Apply виконує детерміновану композицію «вісь – риса-особливість – масштаб злопам'ятності»; методи TermFactor і FocusFactor обчислюють динамічні коефіцієнти для оцінок дипломатичних угод та стратегічного фокуса; RefreshDynamic виконує щоходове згасання настрою й пам'яті.

```
namespace PersonalityMod.Engine
{
    // Per-empire static personality state. Precomputes axis
    multipliers once
    // from the victory vector; Apply does an O(1) read at
    hook time.
    // No static field initializer - instance ctor only (no
    hidden .cctor, project rule).
    internal sealed class PersonalitySnapshot
    {
        internal readonly Archetype Archetype;

        // Sentinel: DynamicTurn holds this until the first
        RefreshDynamic call.
        // const emits no .cctor; static readonly float WOULD.
        internal const int UnrefreshedTurn = int.MinValue;

        // NeutralShare = 1/5 axes: a uniform archetype (all
        0.2) yields mult = 1.0
        // exactly on every axis (passthrough). Structural
        const - not tunable.
        private const float NeutralShare = 0.2f;

        // Precomputed per-axis multipliers, length 5.
        // Index: 0=military, 1=science, 2=diplomacy,
        3=economy, 4=wonder
        // (matches PathMap.Axis* constants).
        private readonly float[] _axisMultiplier;
        private readonly float _floor;
        private readonly float _ceil;
    }
}
```

```

        // Dynamic mood state (Story 2.3 - FR24). The Mood
reference is seeded in ctor and
        // never reassigned (readonly); its contents are
refreshed by the facade under the
        // gate each new turn. DynamicTurn stays mutable so
RefreshDynamic and facade test
        // seams can advance the turn marker directly.
        internal readonly Mood Mood;
        internal int DynamicTurn;

        // Per-rival event memory (Story 3.3 / FR13). Seeded
in ctor alongside Mood.
        internal readonly Memory Memory;

        internal PersonalitySnapshot(Archetype archetype,
                                     float baseline, float
gain,
                                     float floor, float ceil)
        {
            Archetype = archetype;
            _floor      = floor;
            _ceil       = ceil;
            _axisMultiplier = new float[5];

            // Axis order must match PathMap.Axis* constants.
            float[] victories = new float[]
            {
                archetype.VictoryMilitary,    // AxisMilitary
= 0
                archetype.VictoryScience,     // AxisScience
= 1
                archetype.VictoryDiplomacy,   // AxisDiplomacy
= 2
                archetype.VictoryEconomy,     // AxisEconomy
= 3
                archetype.VictoryWonder       // AxisWonder
= 4
            };

            for (int i = 0; i < 5; i++)
            {
                float raw = baseline + gain * (victories[i] -
NeutralShare);
                _axisMultiplier[i] = Clamp(raw, floor, ceil);
            }

            Mood      = new Mood(archetype);
            Memory     = new Memory();
            DynamicTurn = UnrefreshedTurn;
        }

        // O(1) accessor over the precomputed per-axis
multiplier (Story 1.6 telemetry

```

```

        // emphasis-tell read). Returns the PRE-QUIRK axis
multiplier - the raw ctor value
        // from the victory vector only; quirk composition
lives in Apply and is never
        // folded into _axisMultiplier. Out-of-range index →
1f (identity), so a bad axis
        // can never bias a telemetry row. No .ctor, no
alloc.
        internal float GetAxisMultiplier(int axisIndex)
        {
            if (axisIndex < 0 || axisIndex >=
_axisMultiplier.Length) return 1f;
            return _axisMultiplier[axisIndex];
        }

        // O(1) snapshot read: axis branch + quirk branch +
grudge-scale branch, alloc-free.
        // Composition order (DETERMINISTIC): axis → quirk →
grudge-scale.
        // An unmapped, no-quirk, non-grudge-scaled path
returns exact baseValue.
        // NaN/Inf guard belongs at the FACADE boundary
(project-context.md:128);
        // Engine/ stays pure - do not guard here.
        internal float Apply(PathMap map, string path, float
baseValue)
        {
            float result = baseValue;

            if (map.TryGetRule(path, out var axis, out var
inverted))
            {
                float m = _axisMultiplier[axis];

                if (inverted)
                {
                    m = (m != 0f) ? 1f / m : _ceil;
                    m = Clamp(m, _floor, _ceil);
                }

                result = result * m;
            }

            if (map.TryGetQuirkBias(path, out var quirkIndex,
out var bias) && HasQuirk(quirkIndex))
                result = result * bias; // quirk off OR no
quirk row → identity (AC2)

            // Story 3.5b (Task 4): grudge-scale third
multiply for betrayal-attitude paths.
            // Multiplies the quirk-stage output; clamped to
[_floor, _ceil] like the axis stage.
            if (map.IsGrudgeScaled(path))

```

```

        {
            float grudgeFactor = Clamp(1f +
PathMap.GrudgeAttitudeGain * Archetype.Grudge,
                                   _floor, _ceil);
            result = result * grudgeFactor;
        }

        return result;
    }

    // Alloc-free switch - int keys, no enum boxing
(project rule).
    private bool HasQuirk(int quirkIndex)
    {
        switch (quirkIndex)
        {
            case PathMap.QuirkCourtsTheStrong:
return Archetype.CourtsTheStrong;
            case PathMap.QuirkBalancesAgainstLeader:
return Archetype.BalancesAgainstLeader;
            case PathMap.QuirkPunishesBetrayalHarshly:
return Archetype.PunishesBetrayalHarshly;
            case PathMap.QuirkExpandsBeforeFighting:
return Archetype.ExpandsBeforeFighting;
            case PathMap.QuirkExploitsWeakness:
return Archetype.ExploitsWeakness;
            default:
return false;
        }
    }

    // Dynamic per-rival term-value factor (Story 2.4, H2;
extended Story 3.5b).
    // Called by the facade UNDER the gate, after
EnsureRefreshedUnderGate - the Mood is safe
    // to read here. Read-only: no seeding, no allocation
on the H2 hot path.
    // Uses seed-equivalent constants for unseeded rivals
so the global posture deviation
    // reaches H2 even before per-rival emotion seeding
(Epic-3 event path, H3).
    // Belt-and-suspenders negative guard: primary out-of-
range guard is at the facade boundary
    // (rivalIndex < 0 || >= MaxEmpires rejected before
this call).
    // currentTurn (Story 3.5b): threaded to
GrudgeWeightFor for decay elapsed computation.
    internal float TermFactor(int rivalIndex, string
termKind, int currentTurn)
    {
        int kind = MoodTermScale.KindFromString(termKind);
        if (kind == MoodTermScale.KindUnknown) return 1f;

```

```

        if (rivalIndex < 0) return 1f; // belt-and-
suspenders; primary guard is facade-side

        float anger, resent, trust, respect;
        if (Mood.TryGetRival(rivalIndex, out var r))
        {
            anger    = r.Anger;
            resent   = r.Resentment;
            trust    = r.Trust;
            respect  = r.Respect;
        }
        else
        {
            // Seed-equivalent constants: bitwise-
            identical to a freshly seeded RivalMood.
            // DO NOT return 1f here - that would drop the
            global posture deviation in Epic 2
            // where no per-rival slot is ever seeded
            (RefreshDynamic decays existing slots
            // but never seeds; first-touch seeding lands
            on the cold Epic-3 H3 path).
            anger    = 0f;
            resent   = 0f;
            trust    = Archetype.Trustfulness;
            respect  = RivalMood.NeutralBaseline;
        }

        float memGrudge =
        Memory.GrudgeWeightFor(rivalIndex, currentTurn,
        Archetype.Grudge);
        // Alliance veto: only meaningful for the Alliance
        kind (cost-free guard).
        bool everBetrayed = (kind ==
        MoodTermScale.KindAlliance) &&
        Memory.EverBetrayedBy(rivalIndex);

        return MoodTermScale.Factor(kind,
        Mood.MilitaryPosture, Archetype.Agression,
        anger, resent,
        trust, Archetype.Trustfulness,
        respect, RivalMood.NeutralBaseline,
        memGrudge, Archetype.Grudge,
        everBetrayed, Archetype.RiskTolerance); //
2026-06-21 retune: RT-wired War/Peace tilt
    }

    // Dynamic strategy-focus factor (Story 2.5, H2).
    // Called by the facade UNDER the gate, after
    EnsureRefreshedUnderGate - Mood is safe to read.
    // Strategy focus is GLOBAL (per-empire), not per-
    rival: no rivalIndex, no seeding, zero alloc.
    // Non-"PrepareWar" focusKind -> identity (1f); only
    military-build carries the dynamic term this epic.

```

```

        internal float FocusFactor(string focusKind)
        {
            int kind =
StrategyFocusScale.KindFromString(focusKind);
            if (kind == StrategyFocusScale.KindUnknown) return
1f;
            return StrategyFocusScale.Factor(kind,
Mood.MilitaryPosture, Archetype.Aggression);
        }

        // Record a memory event for this empire's snapshot.
        // Seeds the rival emotion slot (cold touch) then
computes sig/weight and appends.
        // Called from the facade handler under the gate (NFR5
/ AC5).
        // instigatorRivalIndex = the empire index of the
instigator
        // targetIndex          = the empire index of the
target (= me, this empire)
        // sig and weight are returned for the instrumentation
row.
        internal void RecordEvent(int turn, int
instigatorRivalIndex, int targetIndex,
MemoryEventType type, out
float sig, out float weight)
        {
            Mood.GetOrSeedRival(instigatorRivalIndex); //
cold-touch seeding
            sig    = EventCatalog.Significance(type);
            weight = EventCatalog.InitialImpact(type,
Archetype);
            // 2026-06-21: make the global DiplomaticOpenness
mood axis dynamic (it had no
            // live caller and sat at the Trustfulness
baseline all game). Sign = event
            // polarity (grievance lowers openness, positive
raises it); magnitude =
            // Significance * gain. Loss is patience-scaled
inside NudgeDiplomaticOpenness;
            // reversion pulls it back. Observability axis
only - it feeds no decision, so
            // game behaviour is unchanged (telemetry col 10
becomes dynamic).
            float opennessSign =
EventCatalog.IsGrievance(type) ? -1f : 1f;
            Mood.NudgeDiplomaticOpenness(opennessSign *
Mood.OpennessEventGain * sig);
            Memory.Add(instigatorRivalIndex,
                new MemoryEntry(turn, instigatorRivalIndex,
targetIndex, type, sig, weight));
        }

```

```

        // Lazy once-per-turn dynamic refresh (FR24). Called
        by the facade under the gate.
        // First touch (DynamicTurn == UnrefreshedTurn): seeds
        the turn marker only
        // no decay or reversion fires (no elapsed turns to
        advance on first contact).
        // Subsequent calls: decay all rivals, revert posture
        by elapsed steps, apply
        // atWar nudge, drift goodwill for peaceful rivals,
        prune+cap live memory.
        // Caller contract: elapsed >= 0 (facade clamps
        silently; negative = no-op here).
        // rivalAtPeace: null when no refresh is due (non-
        stale path passes null and
        // maintenance is skipped); non-null on a stale
        refresh - first-touch also
        // receives a non-null array but the early-return
        above discards it before any
        // maintenance call fires.
        internal void RefreshDynamic(int currentTurn, int
        turnsElapsed, bool atWar,
                                   bool[] rivalAtPeace)
        {
            if (DynamicTurn == UnrefreshedTurn)
            {
                DynamicTurn = currentTurn;
                return; // first touch: seeds turn marker,
advances nothing
            }
            Mood.DecayAllRivals(turnsElapsed);
            for (int i = 0; i < turnsElapsed; i++)
                Mood.RevertPosture();
            if (atWar)
                Mood.NudgeMilitaryPosture(Mood.NudgeMinor);
            // FR18 / AC6: goodwill drift for peaceful rivals
        (Story 3.3).
            if (rivalAtPeace != null)
                Mood.DriftPeacefulRivals(rivalAtPeace);
            // NFR5 / AC7: prune stale and over-cap live
memory entries (Story 3.3).
            Memory.PruneAndCapLive(currentTurn,
            Archetype.Grudge);
            DynamicTurn = currentTurn;
        }

        private static float Clamp(float value, float min,
float max)
        {
            if (value < min) return min;
            if (value > max) return max;
            return value;
        }
    }

```

}

ДОДАТОК Б

Вихідний код класу Mood (динамічний настрій)

Клас Mood реалізує динамічний (швидкий) шар моделі: дві глобальні осі настрою – військову позу та дипломатичну відкритість – і поемоційний стан щодо кожного суперника (гнів, образа, довіра, повага). Настрій реагує на події імпульсами й плавно повертається до базового рівня, заданого архетипом, у проміжках між подіями.

```
using System;
using System.Collections.Generic;

namespace PersonalityMod.Engine
{
    // Per-rival emotion record (FR8) - seven floats seeded
    // from the empire's
    // archetype. Reaction-emotions seed to 0; relationship-
    // emotions seed from
    // Trustfulness / the neutral baseline. Mutable (not
    // readonly): Story 2.2
    // nudges/decays these in place. The seven names are MOOD
    // STATE, not archetype
    // traits - no trait-vocabulary field of its own (AC3).
    // No static field initializer that runs code → no hidden
    // .cctor (project rule);
    // NeutralBaseline is a const (emits no .cctor).
    internal sealed class RivalMood
    {
        // [0,1] scale midpoint - the GDD neutral baseline for
        // Respect/Confidence.
        // Structural const, NOT a <Tunables> knob. Canonical
        // owner is RivalMood
        // (where Respect/Confidence are seeded → unqualified
        // use). A const emits
        // no .cctor; static readonly float WOULD - forbidden.
        internal const float NeutralBaseline = 0.5f;

        // GDD order (gdd.md:140): Fear, Anger, Confidence,
        // Desperation, Trust,
        // Respect, Resentment. Mutable for Story 2.2 in-place
        // dynamics.
        internal float Fear;
        internal float Anger;
        internal float Confidence;
        internal float Desperation;
        internal float Trust;
        internal float Respect;
```

```

        internal float Resentment;

        // Seeding ctor - direct seed copies only; no logic
        (decay/nudge land in 2.2).
        internal RivalMood(Archetype a)
        {
            // reaction-emotions
            Fear = 0f;
            Anger = 0f;
            Desperation = 0f;
            Resentment = 0f;
            // relationship-emotions
            Trust = a.Trustfulness;
            Respect = NeutralBaseline;
            Confidence = NeutralBaseline;
        }
    }

    // Per-empire mood (FR7): two posture axes seeded from the
    archetype + a
    // per-rival emotion store keyed by int rivalIndex.
    MilitaryPosture /
    // DiplomaticOpenness are mood-axis STATE, not trait
    fields (AC3).
    // No static field initializer → no hidden .cctor (project
    rule); the
    // Dictionary is allocated in the instance ctor.
    internal sealed class Mood
    {
        // -- Nudge magnitudes (FR9, AC1). config keys:
        mood.nudge.{minor,moderate,major}
        internal const float NudgeMinor      = 0.05f;
        internal const float NudgeModerate    = 0.10f;
        internal const float NudgeMajor       = 0.20f;

        // Per-event DiplomaticOpenness nudge gain (2026-06-
        21): signedBase = polarity ×
        // OpennessEventGain × event Significance (grievance =
        loss, positive = gain).
        // Makes the global openness axis dynamic on event
        intake; loss is patience-scaled
        // inside NudgeDiplomaticOpenness, reversion pulls it
        back. config key: mood.opennessEventGain
        internal const float OpennessEventGain = 0.5f;

        // -- Reversion rate (FR10, AC2). config key:
        mood.reversionRate
        // Full rate: ReversionRate * (1 -
        _archetype.Stubbornness), computed live.
        internal const float ReversionRate = 0.10f;

        // -- Decay half-lives (FR11, AC3). config keys:
        mood.fearHalfLife,

```

```

        //      mood.grudgeHalfLife.base,
mood.grudgeHalfLife.slope
        internal const float FearHalfLifeTurns      = 10f;
        internal const float HalfLifeBaseTurns      = 5f;
        internal const float HalfLifeEmotionSlope = 35f; // h
= HalfLifeBaseTurns + HalfLifeEmotionSlope * Grudge

        // -- Trust asymmetry factors (AC3). config keys:
mood.trust.buildFactor, mood.trust.dropFactor
        // Gain at half rate; loss at full rate ("builds
slowly, drops sharply").
        internal const float TrustBuildFactor = 0.5f;
        internal const float TrustDropFactor  = 1.0f;

        // Posture axes [0,1] - seeded by direct copy from the
archetype traits.
        internal float MilitaryPosture;
        internal float DiplomaticOpenness;

        // Read-only reference to the owning empire's
archetype, held ONLY to seed
        // posture and new rival slots. The Archetype layer
owns the trait
        // vocabulary; reading the owner is the opposite of
aliasing (AC3) - and it
        // lets GetOrSeedRival seed without the caller
threading the archetype
        // through. Story 2.2 re-reads reversion baselines
live from here, never a
        // copied Mood-stored baseline (single owning layer).
private readonly Archetype _archetype;

        // Per-rival emotion store, keyed by int rivalIndex
(non-boxing default
        // comparer; never an enum or composed-string key -
project rule).
        private readonly Dictionary<int, RivalMood> _rivals;

        // Seeding ctor - posture is a DIRECT COPY of the
archetype traits.
        internal Mood(Archetype a)
        {
            _archetype = a;
            MilitaryPosture = a.Aggression;
            DiplomaticOpenness = a.Trustfulness;
            _rivals = new Dictionary<int, RivalMood>();
        }

        // Seed-on-first-touch: return the existing slot, or
create + store a
        // freshly seeded RivalMood on first touch. Pure
seeding - zero dynamics

```

```

        // (no turn tracking, no refresh, no lock; those are
Story 2.3).
        internal RivalMood GetOrSeedRival(int rivalIndex)
        {
            if (_rivals.TryGetValue(rivalIndex, out var m))
                return m;
            m = new RivalMood(_archetype);
            _rivals[rivalIndex] = m;
            return m;
        }

        // FR10, AC2: one turn's reversion step on both
posture axes toward the live
        // archetype baseline. rate = ReversionRate * (1 -
Stubbornness).
        // Convex step between two [0,1] values stays in
[0,1]; no clamp needed.
        // Per-rival emotions are NOT touched (they decay
separately via DecayRival).
        internal void RevertPosture()
        {
            float rate = ReversionRate * (1f -
_archetype.Stubbornness);
            MilitaryPosture += (_archetype.Aggression -
MilitaryPosture) * rate;
            DiplomaticOpenness += (_archetype.Trustfulness -
DiplomaticOpenness) * rate;
        }

        // FR9, AC1: MilitaryPosture nudge - RAW (no trait
scaling; 2.3 derives the
        // state-based threat source for this channel).
        internal void NudgeMilitaryPosture(float signedBase)
        {
            MilitaryPosture = Clamp01(MilitaryPosture +
signedBase);
        }

        // FR9, AC1: DiplomaticOpenness nudge - loss
(signedBase < 0) scaled by
        // DiplomaticPatienceDecay; gain (signedBase >= 0) is
raw.
        internal void NudgeDiplomaticOpenness(float
signedBase)
        {
            float delta = signedBase < 0f ? signedBase *
_archetype.DiplomaticPatienceDecay : signedBase;
            DiplomaticOpenness = Clamp01(DiplomaticOpenness +
delta);
        }

        // FR9, AC1: per-rival emotion nudges - trait-scaled;
live on Mood (holds

```

```

        // the _archetype reference) so trait scaling reads
        one owner.

        internal void NudgeFear(RivalMood r, float signedBase)
        {
            r.Fear = Clamp01(r.Fear + signedBase *
            _archetype.ThreatOverreaction);
        }

        internal void NudgeAnger(RivalMood r, float
        signedBase)
        {
            r.Anger = Clamp01(r.Anger + signedBase *
            _archetype.Grudge);
        }

        internal void NudgeResentment(RivalMood r, float
        signedBase)
        {
            r.Resentment = Clamp01(r.Resentment + signedBase *
            _archetype.Grudge);
        }

        // FR9, AC3: Trust nudge - asymmetric ("builds slowly,
        drops sharply").
        // Positive signedBase uses TrustBuildFactor; negative
        uses TrustDropFactor.
        internal void NudgeTrust(RivalMood r, float
        signedBase)
        {
            float factor = signedBase >= 0f ? TrustBuildFactor
            : TrustDropFactor;
            r.Trust = Clamp01(r.Trust + signedBase * factor);
        }

        // FR11, AC3: multiplicative half-life decay toward 0
        for Anger, Resentment,
        // Fear only. turnsElapsed is float so the Ring-1 test
        can pass fractional h
        // (e.g. 13.75) and verify exact halving. Caller
        contract: turnsElapsed >= 0.
        internal void DecayRival(RivalMood r, float
        turnsElapsed)
        {
            float hAngerResent = HalfLifeBaseTurns +
            HalfLifeEmotionSlope * _archetype.Grudge;
            float fAngerResent = (float)Math.Pow(0.5,
            turnsElapsed / hAngerResent);
            float fFear          = (float)Math.Pow(0.5,
            turnsElapsed / FearHalfLifeTurns);
            r.Anger              *= fAngerResent;
            r.Resentment         *= fAngerResent;
            r.Fear                *= fFear;
        }

```

```

    }

    // Read-only rival accessor - no seeding, no
allocation; safe on the H2 hot path.
    // Returns false when the slot has not been seeded
(first-touch seeding is GetOrSeedRival,
    // reserved for the cold Epic-3 H3 event-intake path).
Callers handle the unseeded case
    // with seed-equivalent constants so the global
posture deviation still reaches H2.
    internal bool TryGetRival(int rivalIndex, out
RivalMood rival)
    {
        return _rivals.TryGetValue(rivalIndex, out rival);
    }

    // Per-turn bulk decay: iterates every seeded rival
slot.
    // Allocation-OK per project rules: foreach over
Dictionary.Values on a per-turn
    // (not per-call) path, and only for archetyped
empires.
    internal void DecayAllRivals(float turnsElapsed)
    {
        foreach (RivalMood r in _rivals.Values)
            DecayRival(r, turnsElapsed);
    }

    // FR18 (partial, AC4): one peaceful turn's goodwill
rebuild toward baseline.
    // Trust drifts toward archetype Trustfulness; Respect
toward NeutralBaseline.
    // Convex step between two [0,1] values stays in [0,1]
- no clamp needed.
    // Anger/Resentment/Fear NOT touched here - their
relaxation toward 0 is
    // already DecayRival's job (re-decaying here would
double-count them).
    // config key: mood.goodwillDriftRate
    internal const float GoodwillDriftRate = 0.05f;

    internal void DriftGoodwill(RivalMood r)
    {
        // AC6: non-finite state degrades to no drift
rather than NaN-poisoning the slot.
        if (float.IsNaN(r.Trust) ||
float.IsInfinity(r.Trust) ||
            float.IsNaN(r.Respect) ||
float.IsInfinity(r.Respect))
            return;
        r.Trust += (_archetype.Trustfulness - r.Trust)
* GoodwillDriftRate;

```

```

        r.Respect += (RivalMood.NeutralBaseline -
r.Respect) * GoodwillDriftRate;
    }

    // FR18 / AC6 (Story 3.3): drift goodwill for all
seeded rivals that are
    // currently at peace (not at war). rivalAtPeace is
indexed by rivalIndex;
    // indices out of range or false slots are skipped.
Called from RefreshDynamic
    // after the per-turn decay/reversion pass.
    internal void DriftPeacefulRivals(bool[] rivalAtPeace)
    {
        if (rivalAtPeace == null) return;
        foreach (KeyValuePair<int, RivalMood> kv in
_rivals)
        {
            int idx = kv.Key;
            if (idx < 0 || idx >= rivalAtPeace.Length)
continue;
            if (!rivalAtPeace[idx]) continue; // at war →
skip goodwill drift
            DriftGoodwill(kv.Value);
        }
    }

    // [0,1] clamp - mirrors ResponseCurves.Clamp01.
Static method; no .cctor.
    private static float Clamp01(float v)
    {
        return v < 0f ? 0f : (v > 1f ? 1f : v);
    }
}

```

ДОДАТОК В

Вихідний код класу Memory (персистентна пам'ять подій)

Клас Memory реалізує персистентний (тривалий) шар моделі: журнал подій щодо кожного суперника з обчисленням ваги злопам'ятності, експоненційним згасанням у часі та підлогою злопам'ятності, що не дає повністю забути найтяжчі образи. Тут же виконується обрізання застарілих та обмеження кількості активних записів.

```
using System.Collections.Generic;

namespace PersonalityMod.Engine
{
    // Per-empire memory container (FR13/FR19/NFR5). Holds a
    // per-rival log of
    // MemoryEntry values keyed by int rivalIndex (non-boxing
    // default comparer).
    // Bounding (prune + cap) runs after every Add;
    PruneAndCapLive uses compute-on-read
    // live weights for the per-turn maintenance pass (Story
    3.3).
    // No static field initializer - dictionary allocated in
    instance ctor (no .cctor).
    internal sealed class Memory
    {
        // Bounding constants (NFR9 - no magic numbers). Ring-
        1 asserts the exact values.
        internal const float PruneThreshold          = 0.02f;
        internal const int   MaxEntriesPerRival      = 20;
        internal const int   NeverInteractedTurnsSince =
int.MaxValue;

        // Per-rival log. Key = int rivalIndex (non-boxing;
        never an enum or string key).
        private readonly Dictionary<int, List<MemoryEntry>>
        _byRival;

        // Set-once latched facts - permanent regardless of
        decay/prune (decision-log :139).
        // Allocated in instance ctor; no static field
        initializer (no hidden .cctor).
        private readonly HashSet<int> _everBetrayed;
        private readonly HashSet<int> _everAllied;

        internal Memory()
        {
            _byRival          = new Dictionary<int,
List<MemoryEntry>>();
```

```

        _everBetrayed = new HashSet<int>();
        _everAllied   = new HashSet<int>();
    }

    // Append entry and enforce the per-rival invariant
    (≤20, none < 0.02).
    internal void Add(int rivalIndex, MemoryEntry entry)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list))
        {
            list = new List<MemoryEntry>();
            _byRival[rivalIndex] = list;
        }
        list.Add(entry);
        if (EventCatalog.IsBetrayalEvent(entry.Type))
            _everBetrayed.Add(rivalIndex);
        if (entry.Type == MemoryEventType.FormedAlliance)
            _everAllied.Add(rivalIndex);
        Bound(rivalIndex);
    }

    // Enforce bounding invariant for one rival slot.
    // (a) Prune: remove entries with EmotionalWeight
strictly < PruneThreshold.
    // (b) Cap: if still > MaxEntriesPerRival, drop the
lowest-EmotionalWeight
    //         entries. Tie-break: higher Turn wins; if Turn
also ties, earlier
    //         insertion (lower original index) wins -
List<T>.Sort is not stable,
    //         so the insertion index is carried as the final
comparison key.
    internal void Bound(int rivalIndex)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list))
            return;

        // (a) Prune: backward pass to avoid index shift.
        for (int i = list.Count - 1; i >= 0; i--)
        {
            if (list[i].EmotionalWeight < PruneThreshold)
                list.RemoveAt(i);
        }

        // (b) Cap: drop lowest-weight entries until
exactly MaxEntriesPerRival remain.
        if (list.Count > MaxEntriesPerRival)
        {
            // Build a sortable array carrying the
original insertion index.

```

```

        // Sort ascending by (EmotionalWeight, -Turn,
insertionIndex) so the
        // entries to DROP are at the front.
        int count = list.Count;
        int[] order = new int[count];
        for (int i = 0; i < count; i++)
            order[i] = i;

        // Insertion sort - small N ( $\leq$  count-after-
prune, bounded  $\leq$  MaxEntriesPerRival+
        // burst), stable-deterministic without
relying on List.Sort stability.
        for (int i = 1; i < count; i++)
        {
            int key = order[i];
            MemoryEntry ke = list[key];
            int j = i - 1;
            while (j >= 0 && IsLower(list[order[j]],
ke, order[j], key))
            {
                order[j + 1] = order[j];
                j--;
            }
            order[j + 1] = key;
        }

        // order is descending priority:
order[0]=highest, order[count-1]=lowest.
        // Drop the last dropCount entries (lowest
priority).
        int dropCount = list.Count -
MaxEntriesPerRival;
        bool[] drop = new bool[count];
        for (int i = count - dropCount; i < count;
i++)
            drop[order[i]] = true;

        // Remove flagged entries (backward pass).
        for (int i = count - 1; i >= 0; i--)
        {
            if (drop[i]) list.RemoveAt(i);
        }
    }

    // Returns true if entry 'a' (at insertion index ia)
has LOWER priority than
    // entry 'b' (at ib) - i.e. 'a' should be dropped
before 'b' if over-cap.
    // Priority order: higher EmotionalWeight > higher
Turn > lower insertion index.
    private static bool IsLower(MemoryEntry a, MemoryEntry
b, int ia, int ib)

```

```

    {
        if (a.EmotionalWeight != b.EmotionalWeight)
            return a.EmotionalWeight < b.EmotionalWeight;
        if (a.Turn != b.Turn)
            return a.Turn < b.Turn;    // higher Turn =
higher priority
        return ia > ib;                // lower insertion
index = higher priority
    }

    // Per-turn maintenance pass (Story 3.3 / AC7). Uses
compute-on-read live
    // weights so stored peaks remain immutable (AC4). For
each rival slot:
    // (a) NaN guard: corrupted peaks become 0 rather
than surviving forever.
    // (b) Prune: drop entries whose live weight <
PruneThreshold.
    // (c) Cap: drop lowest-live-weight entries until ≤
MaxEntriesPerRival.
    // Caller contract: currentTurn ≥ entry.Turn for all
entries (no negative elapsed).
    internal void PruneAndCapLive(int currentTurn, float
grudge)
    {
        // Snapshot keys to avoid modifying the dictionary
while iterating.
        int[] keys = new int[_byRival.Count];
        int ki = 0;
        foreach (int k in _byRival.Keys)
            keys[ki++] = k;

        for (int kk = 0; kk < ki; kk++)
        {
            int rivalIndex = keys[kk];
            if (!_byRival.TryGetValue(rivalIndex, out var
list)) continue;

            // (a+b) Compute live weights and prune:
backward pass.
            for (int i = list.Count - 1; i ≥ 0; i--)
            {
                var e = list[i];
                float elapsed = currentTurn - e.Turn;
                if (elapsed < 0f) elapsed = 0f;
                float w = EventCatalog.LiveWeight(e.Type,
e.EmotionalWeight, grudge, elapsed);
                if (float.IsNaN(w) || float.IsInfinity(w))
w = 0f; // L2 advisory guard
                if (w < PruneThreshold)
                    list.RemoveAt(i);
            }
        }
    }

```

```

// (c) Cap by live weight: same insertion-sort
logic as Bound().
if (list.Count > MaxEntriesPerRival)
{
    // Build live-weight array for each
surviving entry.
    int count = list.Count;
    float[] liveW = new float[count];
    for (int i = 0; i < count; i++)
    {
        var e = list[i];
        float elapsed = currentTurn - e.Turn;
        if (elapsed < 0f) elapsed = 0f;
        float w =
EventCatalog.LiveWeight(e.Type, e.EmotionalWeight, grudge,
elapsed);
        if (float.IsNaN(w) ||
float.IsInfinity(w)) w = 0f;
        liveW[i] = w;
    }

    int[] order = new int[count];
    for (int i = 0; i < count; i++) order[i] =
i;

    // Insertion sort descending by (liveW, -
Turn, insertionIndex) → drop-last at tail.
    for (int i = 1; i < count; i++)
    {
        int key = order[i];
        float keyW = liveW[key];
        MemoryEntry ke = list[key];
        int j = i - 1;
        while (j >= 0 &&
IsLowerLive(liveW[order[j]], list[order[j]], keyW, ke,
order[j], key))
        {
            order[j + 1] = order[j];
            j--;
        }
        order[j + 1] = key;
    }

    int dropCount = count -
MaxEntriesPerRival;
    bool[] drop = new bool[count];
    for (int i = count - dropCount; i < count;
i++)
        drop[order[i]] = true;

    for (int i = count - 1; i >= 0; i--)
        if (drop[i]) list.RemoveAt(i);
}

```

```

        }
    }

    // Lower-priority predicate for PruneAndCapLive cap
    sort (live-weight variant).
    private static bool IsLowerLive(float aW, MemoryEntry
a, float bW, MemoryEntry b, int ia, int ib)
    {
        if (aW != bW)        return aW < bW;
        if (a.Turn != b.Turn) return a.Turn < b.Turn;
        return ia > ib;
    }

    // 0 if the rival slot has not been seeded.
    internal int CountFor(int rivalIndex)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list))
            return 0;
        return list.Count;
    }

    // By-value read (struct copies on return - caller
cannot mutate the stored entry).
    internal MemoryEntry EntryAt(int rivalIndex, int
index)
    {
        return _byRival[rivalIndex][index];
    }

    // --- Summary queries (Story 3.4 / FR20) ---
    // Read the live list directly (post-PruneAndCapLive
pass).  $O(\leq 20)$  = bounded-constant.
    // for-index only: foreach over List<MemoryEntry> as
IEnumerable boxes the enumerator on Mono 2.x.
    // Latched flags read HashSet<int>.Contains - non-
boxing default int comparer, no allocation.

    internal int TimesAttackedBy(int rivalIndex)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list)) return 0;
        int n = 0;
        for (int i = 0; i < list.Count; i++)
            if (list[i].Type ==
MemoryEventType.AttackedMyArmy) n++;
        return n;
    }

    internal int TreatiesBrokenBy(int rivalIndex)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list)) return 0;

```

```

        int n = 0;
        for (int i = 0; i < list.Count; i++)
            if
(EventCatalog.IsBetrayalEvent(list[i].Type)) n++;
        return n;
    }

    internal int TimesHelpedBy(int rivalIndex)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list)) return 0;
        int n = 0;
        for (int i = 0; i < list.Count; i++)
        {
            MemoryEventType t = list[i].Type;
            if (t == MemoryEventType.DeclaredWarOnMyEnemy
|| t == MemoryEventType.GaveGift) n++;
        }
        return n;
    }

    internal bool EverBetrayedBy(int rivalIndex)
    {
        return _everBetrayed.Contains(rivalIndex);
    }

    internal bool EverAlliedWith(int rivalIndex)
    {
        return _everAllied.Contains(rivalIndex);
    }

    // currentTurn - max(Turn) over all live entries;
    NeverInteractedTurnsSince when empty.
    // Iterates to find max: cap-drop can remove a high-
    turn low-weight entry, so
    // list[Count-1] is not guaranteed to be the latest
    entry by turn.
    internal int TurnsSinceLastInteractionWith(int
rivalIndex, int currentTurn)
    {
        if (!_byRival.TryGetValue(rivalIndex, out var
list) || list.Count == 0)
            return NeverInteractedTurnsSince;
        int maxTurn = list[0].Turn;
        for (int i = 1; i < list.Count; i++)
            if (list[i].Turn > maxTurn) maxTurn =
list[i].Turn;
        int d = currentTurn - maxTurn;
        return d < 0 ? 0 : d;
    }

    // Story 3.5b / FR20: decayed live grudge weight
    toward rivalIndex.

```

```

        // Aggregate = MAX live grievance weight over the
rival's entries (not sum).
        // [0,1] contract enforced by final clamp - LiveWeight
at t=0 = storedPeak
        // can exceed 1.0 for events like CapturedMyCity
(Sig*scale may be >1.0).
        // O(≤20), alloc-free, for-index (no foreach-over-
IEnumerable boxing on net35).
        // Caller contract: currentTurn >= entry.Turn; non-
finite intermediate → 0 (advisory).
        internal float GrudgeWeightFor(int rivalIndex, int
currentTurn, float grudge)
        {
            if (!_byRival.TryGetValue(rivalIndex, out var
list)) return 0f;
            float maxW = 0f;
            for (int i = 0; i < list.Count; i++)
            {
                MemoryEntry e = list[i];
                if (!EventCatalog.IsGrievance(e.Type))
continue;
                float elapsed = currentTurn - e.Turn;
                if (elapsed < 0f) elapsed = 0f;
                float w = EventCatalog.LiveWeight(e.Type,
e.EmotionalWeight, grudge, elapsed);
                if (float.IsNaN(w) || float.IsInfinity(w)) w =
0f;
                if (w > maxW) maxW = w;
            }
            if (maxW > 1f) maxW = 1f;
            if (maxW < 0f) maxW = 0f;
            return maxW;
        }

        // Story 3.6 / FR27: maximum decayed grudge weight
this empire holds against ANY
        // single rival at currentTurn - the "grudge vs top
rival" signal logged as
        // grudgeWeightVsTopRival. Reuses GrudgeWeightFor per
rival slot (already [0,1]);
        // returns 0f when no rival slot holds a grievance.
Per-TURN path (EmitPostureRow,
        // once per empire per turn) - modest-alloc license
applies; the Dictionary<int,...>
        // foreach uses the struct enumerator (no boxing -
Mono trap avoided). No mutation.
        internal float MaxGrudgeWeight(int currentTurn, float
grudge)
        {
            float max = 0f;
            foreach (KeyValuePair<int, List<MemoryEntry>> kv
in _byRival)
            {

```

```
float w = GrudgeWeightFor(kv.Key, currentTurn,
grudge);
    if (w > max) max = w;
    }
    return max; // already in [0,1] (GrudgeWeightFor
clamps at the query boundary)
    }
}
```

ДОДАТОК Г

Вихідний код класу ResponseCurves (криві відгуку)

Клас ResponseCurves містить чотири криві відгуку, що перетворюють входи рішень у нормовані оцінки корисності: лінійну, квадратичну, логістичну та експоненційне згасання, а також обчислення середини логістичної кривої з ризикової толерантності архетипу.

```
using System;

namespace PersonalityMod.Engine
{
    // Pure static math - no instance state, no EL/Unity
    // types, no allocation.
    // Runs on the per-call evaluation path once Story 1.4
    // wires it through the facade.
    internal static class ResponseCurves
    {
        // curve.quadraticExponent - AR9 resolved: exponent 2
        internal const float DefaultQuadraticExponent = 2f;

        // curve.expDecayLambda - AR9 resolved:  $\lambda$  1.0
        internal const float DefaultExpDecayLambda = 1f;

        // curve.logisticK - runtime tunable (gdd.md:245), not
        // source-pinned; default 2.0
        internal const float DefaultLogisticK = 2f;

        private static float Clamp01(float v)
        {
            return v < 0f ? 0f : (v > 1f ? 1f : v);
        }

        internal static float Linear(float x)
        {
            return Clamp01(x);
        }

        internal static float Quadratic(float x, float
        exponent)
        {
            return Clamp01((float)Math.Pow(x, exponent));
        }

        // Logistic is (0,1) for all finite x; no clamp
        // needed.
        // Large  $\pm x$  saturates to  $\sim 1/\sim 0$  without NaN/Inf.
        internal static float Logistic(float x, float m, float
        k)
```

```

    {
        return 1f / (1f + (float)Math.Exp(-k * (x - m)));
    }

    // War-curve midpoint:  $m = 1.5 - 0.5 \cdot \text{RiskTolerance}$ 
    (GDD $Response curves).
    // Structural constant derived from RiskTolerance; not
    a config tunable.
    internal static float LogisticMidpoint(float
riskTolerance)
    {
        return 1.5f - 0.5f * riskTolerance;
    }

    // Valid domain:  $x \geq 0$  (output  $\in (0,1]$ ). Negative  $x$ 
    grows  $>1$  by definition;
    // callers stay in-domain; facade boundary guards any
    non-finite output.
    internal static float ExpDecay(float x, float lambda)
    {
        return (float)Math.Exp(-lambda * x);
    }
}

```

ДОДАТОК Д

Вихідний код класу WarUtility (корисність оголошення війни)

Клас WarUtility демонструє типове рішення на основі корисності: обчислення схильності до оголошення війни як зваженого середнього трьох міркувань (перевага в силі, військовий нахил, можливість) із подальшим множенням на стримувальний коефіцієнт союзу. Ваги міркувань залежать від рис архетипу.

```
namespace PersonalityMod.Engine
{
    // Archetype-only declare-war utility (Story 1.4,
    FR4/FR6).
    // BCL-only - no EL/Unity types, no
    FaultGuard/Config/Facade references.
    // No static field initializers (no .cctor) - const
    defaults only.
    // Runs on the H2 hot path once Epic 2 wires it; no alloc,
    no LINQ, no boxing.
    internal static class WarUtility
    {
        // config key: war.militaryMoodWeight - GDD fixed 0.8
        (gdd.md:228)
        internal const float MilitaryMoodWeight = 0.8f;

        // config key: war.powerWeightBase - rule 0.5 + 0.5·(1
        - RiskTolerance) (gdd.md:227)
        internal const float PowerWeightBase = 0.5f;

        // config key: war.powerWeightRiskGain
        internal const float PowerWeightRiskGain = 0.5f;

        // config key: war.opportunityWeightFloor - the ':0.1'
        branch when ExploitsWeakness is off (gdd.md:230)
        internal const float OpportunityWeightFloor = 0.1f;

        // config key: war.allyRestraintHonored - suppressor k
        when HonorsAlliances=true (gdd.md:231)
        internal const float AllyRestraintHonored = 0.8f;

        // config key: war.allyRestraintDefault - suppressor k
        when HonorsAlliances=false
        internal const float AllyRestraintDefault = 0.2f;

        // Weight for the power-advantage consideration.
        // Cautious archetypes (low RiskTolerance) weight the
        odds more.
        internal static float PowerWeight(Archetype a)
```

```

        {
            return PowerWeightBase + PowerWeightRiskGain * (1f
- a.RiskTolerance);
        }

        // Weight for the military-lean consideration (fixed
GDD value).
        internal static float MilitaryWeight()
        {
            return MilitaryMoodWeight;
        }

        // Weight for the opportunity consideration (FR6 war-
relevant quirk: ExploitsWeakness).
        internal static float OpportunityWeight(Archetype a)
        {
            return a.ExploitsWeakness ? (1f -
a.OpportunismThreshold) : OpportunityWeightFloor;
        }

        // Multiplicative ally-restraint suppressor (FR6 war-
relevant quirk: HonorsAlliances).
        // bond ∈ [0,1] (current trust/alliance strength).
bond=0 → factor=1 (no effect).
        internal static float AllyRestraintFactor(Archetype a,
float bond)
        {
            return 1f - (a.HonorsAlliances ?
AllyRestraintHonored : AllyRestraintDefault) * bond;
        }

        // Archetype-only declare-war utility (3
considerations + ally-restraint suppressor).
        // Grudge/mood terms are Epic 2/3 - NOT referenced
here (no Mood/Memory types).
        //
        // powerRatio = myMil/theirMil (> 0)
        // opportunity ∈ [0,1] - rival distracted / at war
elsewhere
        // bond ∈ [0,1] - current trust/alliance strength
        // logisticK - steepness of the power-advantage
logistic curve (DefaultLogisticK=2f)
        //
        // Out-of-domain inputs are the caller's contract.
        // NaN/Inf backstop is the facade boundary (Epic 2
wiring) - Engine/ stays pure.
        internal static float DeclareWarUtility(Archetype a,
float powerRatio,
float
opportunity, float bond, float logisticK)
        {
            float m =
ResponseCurves.LogisticMidpoint(a.RiskTolerance);

```

```

        float v1 = ResponseCurves.Logistic(powerRatio, m,
logisticK); // power advantage
        float v2 = a.Aggression;
// military lean (Epic-1 stand-in for MilitaryPosture)
        float v3 = ResponseCurves.Linear(opportunity);
// opportunity

        float w1 = PowerWeight(a);
        float w2 = MilitaryWeight();
        float w3 = OpportunityWeight(a);

        // Inline weighted average (alloc-free: no float[]
for a fixed 3-term sum).
        // w1 >= 0.5f, w2 = 0.8f always > 0; w3 can be 0
when ExploitsWeakness && OT=1.0.
        // Divide-guard is defensive - NaN is invisible to
the fail-safe (project-context.md:128).
        var sumWv = w1 * v1 + w2 * v2 + w3 * v3;
        var sumW  = w1 + w2 + w3;
        var u      = sumW <= 0f ? 0f : sumWv / sumW;

        // Multiplicative ally-restraint suppressor
(gdd.md:231).
        u *= AllyRestraintFactor(a, bond);

        return u;
    }
}

```

ДОДАТОК Е

Відомість кваліфікаційної роботи

Позначення					Найменування		Дод. відомості	
					Текстові документи			
1.					Пояснювальна записка		116 с.	