Test Project

Session 4 – GreenWave

IT Software Solutions for Business

Submitted by:
Bence Lakos - Ingenimind Kft, Independent Test Project Designer
Benjamin Frost, Independent Test Project Validator

WSC2026_TP09_en

Contents

Introduction **3**
Description of project and tasks **3**
Global conventions (apply to every component) 3
Components 4
Component A — ISignalProgramValidator *4*
Component B — IDemandCalculator *4*
Component C — ISignalOptimizer *5*
Instructions to the Competitor **7**
Other **7**
Appendix — Domain model **7**

© WorldSkills International

Introduction

GreenWave is a non-profit urban-mobility initiative upgrading a city corridor of signalized intersections from old fixed-time signals to smarter control. Before anything adaptive can be built, the city needs a **correct, deterministic fixed-time timing core**: given an intersection's layout and its measured traffic demand, compute the **optimal cycle length and green split** using **Webster's method** — the textbook fixed-time optimization.

In this session you build that core. Later sessions reuse and extend it.

Description of project and tasks

Implement these three interfaces from the supplied **GreenWave.Control.Contracts** package, in GreenWave.Control.Services:

1. **ISignalProgramValidator** — validate an intersection's signal program.
2. **IDemandCalculator** — turn detector readings into per-phase demand ratios.
3. **ISignalOptimizer** — **Webster's method**: compute the cycle length and green-time split.

They form a pipeline: *validate the program → measure demand → optimize the timing.* All three are pure, deterministic, and headless — no UI, database, clock, randomness, or I/O. All formulas are given below, so **no traffic-engineering background is required**.

Global conventions (apply to every component)

1. **Determinism.** No randomness, clock, culture-dependent parsing, or I/O. Same input ⇒ same output.
2. **Never throw.** No method may throw for any input. On invalid/degenerate input, return the **documented sentinel**. Throwing fails the relevant tests.
3. **Units.** Times are **whole seconds**; flows are **vehicles per hour (vph)**.
4. **Rounding.** Round half away from zero (MidpointRounding.AwayFromZero). The *demand ratios* computed in Component B — each movement's flow ÷ saturation, and a phase's critical ratio — are kept to **4 decimal places**; cycle length and green times are **whole seconds**.
5. **Collections.** A null collection argument is treated as empty; never return null for a collection. **Preserve input (phase/movement) order** in outputs unless told to sort.
6. **Identifiers.** All Id comparisons are **case-sensitive ordinal** string matches.
7. **Marking tolerance.** Integer outputs (green times, cycle length, issue codes) are compared **exactly**.

© WorldSkills International

Components

Component A — ISignalProgramValidator

```
ValidationResult Validate(SignalProgram program);
```

Return an ordered list of **issue codes**. The program is valid (IsValid = true, empty Issues) only if it passes every check. Evaluate the checks **in this order**, appending each code **the first time** its rule is violated (no duplicates):

Order	Issue code	Condition
1	empty-program	program is null, or Phases is null/empty
2	unknown-movement	a phase references a MovementId not present in Movements
3	conflicting-movements	a single phase contains two movements listed as a Conflict pair
4	invalid-green-bounds	any phase has MinGreen <= 0, MaxGreen <= 0, or MinGreen > MaxGreen
5	invalid-cycle-bounds	MinCycle <= 0, MaxCycle <= 0, or MinCycle > MaxCycle
6	missing-pedestrian-walk	a phase with IsPedestrian == true has PedestrianWalk null or <= 0

- IsValid is true only when Issues is empty.
- A null program or null inner collections ⇒ { IsValid:false, Issues:["empty-program"] }.
- **Sentinel:** never throw; always return a ValidationResult.

Example. A program whose phase references undefined movement M9, and whose phase P1 has MinGreen 10, MaxGreen 5, returns IsValid:false, Issues:["unknown-movement","invalid-green-bounds"].

Component B — IDemandCalculator

```
IReadOnlyList<PhaseDemand> Calculate(SignalProgram program, IReadOnlyList<DetectorFrame> frames);
```

Compute each phase's **critical flow ratio** y_i — the input Webster needs, where:

- $a_{m,f}$ — the arrivals reported for movement m in frame f (vph).
- k — the number of frames that report movement m.
- q_m — movement m's mean arrival flow (vph).
- s_m — movement m's saturation flow (vph), from the program.
- y_i — phase i's critical flow ratio.

Then three steps.

© WorldSkills International

Aggregate arrivals. A movement's arrival flow (vph) is the **mean** of its readings across the frames that report it — the sum of its readings $a_{m,f}$ over the number k of reporting frames, to 4 dp. A movement absent from every frame has arrival flow `0`:

$$q_m = \frac{\sum_{f=1}^{k} a_{m,f}}{k}$$

Movement ratio. A movement's demand ratio is its arrival flow divided by its saturation flow, to 4 dp. If $s_m \le 0$ the ratio is `0` (documented — this avoids a divide-by-zero):

$$\frac{q_m}{s_m}$$

Phase critical ratio. A phase's critical ratio y_i is the **largest** movement ratio among the movements m in that phase, to 4 dp. A phase with no movements has $y_i = 0$:

$$y_i = \max_{m \in phase\ i} \frac{q_m}{s_m}$$

Return one `PhaseDemand` per phase, **in program phase order**.

- **Sentinel:** null/empty `frames` ⇒ every phase $y_i = 0$. Never throw.

Example. Phase `P1` has two movements: (540 vph, saturation 1800) and (300 vph, saturation 1900). Their ratios are `0.3000` and `0.1579`, so the phase's critical ratio is `0.3000`.

Component C — `ISignalOptimizer`

```
PhaseAllocation Optimize(SignalProgram program, IReadOnlyList<PhaseDemand> demand);
```

Compute the **optimal cycle length and green split**. The symbols used below:

- n — the number of phases.
- ℓ — the per-phase lost time (`LostTimePerPhase`), in seconds.
- L — the total lost time, $L = n \cdot \ell$.
- y_i — phase i's critical flow ratio (from Component B).
- Y — the sum of all phases' critical ratios.
- C_0 — Webster's optimal cycle length (seconds).
- C — the chosen cycle length, clamped to `[MinCycle, MaxCycle]`.
- G — the total effective green to share out, $G = C - L$.
- g_i — phase i's green time (seconds).
- R — the seconds left to distribute after flooring.

Every formula is given below — first in words, then written out.

Derived values. Let n be the number of phases and ℓ the per-phase lost time (`LostTimePerPhase`); the total lost time is the per-phase lost time times the number of phases:

$$L = n \cdot \ell$$

© WorldSkills International

and Y is the sum of the phases' critical ratios y_i (a phase missing from demand counts as $y_i = 0$):

$$Y = \sum_{i=1}^{n} y_i$$

Step 1 — optimal cycle length. If $Y \geq 1$ the intersection is saturated, so use the longest legal cycle, $C = MaxCycle$. Otherwise take Webster's optimum, round it to a whole second (away-from-zero), then clamp to [MinCycle, MaxCycle] to get C:

$$C_0 = \frac{1.5L + 5}{1 - Y}$$

Step 2 — total effective green. The green available to share out is the cycle minus the total lost time:

$$G = C - L$$

Step 3 — proportional split. Each phase's raw green is its share of G in proportion to its critical ratio. (If $Y = 0$ there is no demand, so split G equally instead: $g_i = G/n$.)

$$g_i = \frac{y_i}{Y} \cdot G$$

Step 4 — round, conserving the total. Take the integer part (floor) of each g_i; the seconds left over are handed out one at a time to the phases with the largest fractional remainders, ties broken by lowest phase index:

$$R = G - \sum_{i=1}^{n} \lfloor g_i \rfloor$$

Step 5 — apply green bounds. Clamp each phase's green to [MinGreen, MaxGreen] (do **not** redistribute afterwards). The realized cycle reported back is the lost time plus the actual greens:

$$CycleLength = L + \sum_{i=1}^{n} g_i$$

- Output in **program phase order**. **Sentinel:** null/empty Phases ⇒ { CycleLength: 0, Greens: [] }. Never throw.

Example. Three phases, LostTimePerPhase = 4 (L = 12), MinCycle 40, MaxCycle 120, generous green bounds. y = [0.30, 0.25, 0.20] ⇒ Y = 0.75.

- $C_0 = (1.5 \cdot 12 + 5)/(1 - 0.75) = 23/0.25 = 92$ ⇒ C = 92.
- $G = 92 - 12 = 80$.
- raw g = [32.000, 26.667, 21.333]; floors [32, 26, 21] = 79; R = 1 → largest remainder is phase 2 (.667) ⇒ greens [32, 27, 21].
- All within bounds ⇒ CycleLength = 12 + 80 = 92, Greens = [32, 27, 21].

(The optimizer's output is exactly what the Simulator animates and what its Webster-baseline delay/LOS readout is computed from — a quick way to sanity-check your numbers.)

© WorldSkills International

Instructions to the Competitor

This section contains the instructions given to the Competitor for this session. The instructions can be increased and improved at the Competition during the preparation time.

Other

Use this section for information that is important but is not covered in the previous sections.

Appendix — Domain model

The three interfaces operate on these immutable `record` types from `GreenWave.Control.Contracts` (field names are illustrative of the supplied types).

`Movement` — one permitted vehicle movement.

Field	Type	Description
`Id`	`string`	Unique movement identifier.
`Turn`	`TurnType`	Direction — one of `Through`, `Left`, `Right`.
`SaturationFlow`	`int`	Saturation (capacity) flow, in vehicles per hour.

`Phase` — the movements that receive a green signal together.

Field	Type	Description
`Id`	`string`	Unique phase identifier.
`MovementIds`	`IReadOnlyList<string>`	Ids of the movements this phase serves.
`IsPedestrian`	`bool`	True when this is a pedestrian phase.
`MinGreen`	`int`	Minimum green time, in seconds.
`MaxGreen`	`int`	Maximum green time, in seconds.
`PedestrianWalk`	`int?`	Walk time in seconds; set only when `IsPedestrian`.

`Conflict` — a movement pair that may not be green at the same time.

Field	Type	Description
`MovementA`	`string`	First movement id of the conflicting pair.
`MovementB`	`string`	Second movement id of the conflicting pair.

`SignalProgram` — the intersection configuration. Input to Components A, B and C.

Field	Type	Description
`Movements`	`IReadOnlyList<Movement>`	Every movement at the intersection.
`Phases`	`IReadOnlyList<Phase>`	The phases, in program order.
`LostTimePerPhase`	`int`	Lost time charged per phase, in seconds.

© WorldSkills International

Field	Type	Description
MinCycle	int	Minimum legal cycle length, in seconds.
MaxCycle	int	Maximum legal cycle length, in seconds.
Conflicts	IReadOnlyList<Conflict>	Movement pairs that may not run together.

DetectorFrame — one snapshot of measured demand. Input to Component B.

Field	Type	Description
TimeSecond	int	The second at which the frame was sampled.
ArrivalsByMovement	IReadOnlyDictionary<string, int>	Arrival flow (vehicles per hour) keyed by movement id.

PhaseDemand — a phase's critical flow ratio. Output of Component B, input to Component C.

Field	Type	Description
PhaseId	string	The phase this demand belongs to.
CriticalRatio	double	The phase's critical flow ratio, to 4 decimal places.

PhaseAllocation — the computed timing. Output of Component C.

Field	Type	Description
CycleLength	int	The realized cycle length, in seconds.
Greens	IReadOnlyList<PhaseGreen>	The green time allocated to each phase.

PhaseGreen — one phase's green time within a PhaseAllocation.

Field	Type	Description
PhaseId	string	The phase.
Green	int	Its green time, in seconds.

ValidationResult — validity and the ordered issue codes. Output of Component A.

Field	Type	Description
IsValid	bool	True only when Issues is empty.
Issues	IReadOnlyList<string>	The issue codes, in the order they were raised.

© WorldSkills International