Modeling Entity Movement

Quiz Study Guide — Chapter 8

Manuel D. Rossetti, PhD P.E.

How to Use This Deck

This deck is a **checklist**, not a summary. Every slide is phrased as something you should be able to *state* or *recognize* without looking it up.

- **Question formats:** multiple choice, true/false, matching, fill-in-the-blank, multi-slot fill-in-the-blank.
- The **Common Traps** slides are built from the actual wrong answers — read them last.

The through-line

Movement can be free ("entities with little feet"), **resource-constrained** (a worker or vehicle must be available), or **space-constrained** (a conveyor with a finite number of cells).

What This Chapter Covers

Spatial movement

- Spatial models and `DistancesModel`
- `MovableResource` and pools
- Selection and allocation rules
- `move`, `moveWith`, `transportWith`

Conveyors

- Segments, cells, velocity, type
- Accumulating versus non-accumulating
- `requestConveyor`, `rideConveyor`, `exitConveyor`
- `convey` and `transferTo`

Objectives

You should be able to:

- Contrast "little feet" movement with resource-constrained movement
- State the two abstract members a `SpatialModel` subclass must implement
- Build a `DistancesModel` and attach it to a process model
- Describe `MovableResource` and name the default allocation rule
- Write the movable-resource transport pattern and its one-call shortcut
- Describe conveyor structure, the two types, and the three ride functions
- Explain the seize and release **order** in the resource-constrained test and repair model

Little Feet versus Constrained Movement

With **"entities with little feet"** you model movement with `delay()`. An **infinite supply of movers is implicitly assumed** and only the move-time matters.

With **resource-constrained movement**, a finite resource — a worker, a vehicle — may itself be unavailable, so the part may **wait** for a mover.

Resource-Constrained Test and Repair

Three `ResourcePoolWithQ` instances:

- `diagnosticWorkers` — DW1, DW2
- `repairWorkers` — RW1, RW2, RW3
- `transportWorkers` — **all eight** workers: DW1, DW2, TW1, TW2, TW3, RW1, RW2, RW3

So each worker appears both in its station's pool and in the transport pool.

Order matters at a station:

- **Seize the worker BEFORE the machine** — the worker is needed to operate it.
- **Release the machine BEFORE the worker** — the worker is needed to finish it.

`seize(transportWorkers, seizePriority = KSLEvent.MEDIUM_PRIORITY)` gives parts waiting **for processing** (default, higher priority) precedence over parts merely waiting **for transport**.

Spatial Models

A **spatial model** represents the physical space: it provides the distance between locations and a `defaultLocation`.

A subclass must implement, besides `defaultLocation`:

- `distance(fromLocation, toLocation): Double`
- `compareLocations(firstLocation, secondLocation): Boolean`

Concrete model	Distance basis
`DistancesModel`	An **origin/destination matrix** — the simplest
Great Circle	GPS coordinates with a **circuity factor**
Euclidean 2D Plane	Straight-line distance between 2D points
Rectangular Grid	Rows $\times$ columns of cells starting at $(0, 0)$

Using a DistancesModel

```
val station1 = dm.Location("Station1")          // inner class of DistancesModel
dm.addDistance(fromLoc, toLoc, distance, symmetric = true)
spatialModel = dm                                // make it active for entities
defaultVelocity = ...                            // default movement velocity
```

- Locations are instances of the **inner `Location` class** of `DistancesModel`, and implement `LocationIfc`.
- Distances **may be asymmetric**; `addDistance` defaults `symmetric` to **true**.
- An `Entity` implements `SpatialElementIfc` and has `initialLocation`, `previousLocation`, `currentLocation`, `isMoving`, and `velocity` — but **no `nextLocation`**.

Movable Resources

Class	Role
`MovableResource`	Subclass of `Resource` also implementing `SpatialElementIfc` and `VelocityIfc`
`MovableResourceWithQ`	Constructed as `(parent, initialLocation, velocityRV, name)`
`MovableResourcePoolWithQ`	A "motor pool" of movable resources with a shared request queue
`MovableResourceSelectionRuleIfc`	Default returns **all currently available** movable resources
`MovableResourceAllocationRuleIfc`	Picks **one** from the eligible list, given the request's location

Rules and Statistics

Default allocation rule: `ClosestMovableResourceAllocationRule` — allocate the mover nearest the request's location.

Rule	When you would choose it
`Furthest...`	Return movers from the **outskirts** of the spatial model to more central activities
`Random...`	Pick uniformly among the available movers
`LeastUtilized...`	Balance load by utilization
`LeastSeized...`	Balance load by seize count

Per movable resource the KSL adds `FracTimeMoving`, `FracTimeTransporting`, and `FracTimeMovingEmpty` to the ordinary resource statistics.

The Transport Pattern

```
val a = seize(mover)
move(mover, entity.currentLocation, emptyVelocity)   // deadhead to the entity
delay(loadingDelay)
moveWith(mover, toLoc, transportVelocity)            // travel together
delay(unLoadingDelay)
release(a)
```

- `moveWith` requires the entity and the movable resource to be at the **same location** — it moves them together.
- `transportWith(mover, toLoc, ...)` wraps that whole pattern into **one call**, defaulting to zero load/unload delays and the resource's default velocity. It does **not** require a manual `seize()` first.

Conveyors

A **`Conveyor`** is a `ModelElement` made of a series of **`Segments`**, each with a **`String`** origin and destination, divided into equal-size **cells** of length `cellSize`, moving at a **single velocity**, with type **`ACCUMULATING`** or **`NON_ACCUMULATING`**.

- Conveyor locations are **`Strings`** — conveyors do **not** use a spatial model, though the strings may match the names of `LocationIfc` instances elsewhere.
- Cells are numbered **from 1** at the entry of the first segment to n at the exit of the last.
- The maximum cells any one entity may occupy is set by the builder's **`maxCellsAllowed`**.
- A conveyor is **circular** when the entry location of the first segment equals the exit location of the last.
- The conveyor knows which locations are **downstream**: a ride request must be feasible along the conveyor's directed path.

Accumulating versus Non-Accumulating

`ACCUMULATING`

The conveyor **keeps running**. Items behind a blockage continue forward until they cannot advance, **queueing on the conveyor itself**.

`NON_ACCUMULATING`

The **entire conveyor disengages** when an item enters at an entry cell or stops at a destination. All items stop; spacing stays constant while moving.

In the "work on the conveyor" variant, the **accumulating** case shows near-zero queue lengths **at the workers** — the queue physically lives on the conveyor as occupied cells. The **non-accumulating** case shows a **severe bottleneck at the entry access queue**, because the whole conveyor stops during processing.

The Conveyor Builder

```
Conveyor.builder(this, "Conveyor")
    .conveyorType(Conveyor.Type.NON_ACCUMULATING)
    .velocity(30.0)
    .cellSize(1)
    .maxCellsAllowed(1)
    .firstSegment(enter, station1, 70)
    .nextSegment(station2, 40)
    .nextSegment(exit, 60)
    .build()
```

For plan-dependent space, use a **`mapOf(testPlan to cellsNeeded)`** and pass `numCellsNeeded = cellsNeeded` to `convey(...)`, with `maxCellsAllowed = 2` on the conveyor.

Riding a Conveyor

Function	Effect
`requestConveyor(conveyor, entryLocation, numCellsNeeded)`	Allocates cells and returns a `ConveyorRequestIfc` — the "ticket to ride". The entity **holds** the cells (blocking the entry cell) but is **not yet riding**.
`rideConveyor(conveyorRequest, destination)`	Occupies the cells and moves to the destination; updates `currentLocation` if the destination implements `LocationIfc`
`exitConveyor(conveyorRequest)`	Moves through the occupied cells — **a real delay** — then deallocates them

Two Convenience Functions

Function	Effect
`convey(conveyor, entry, destination, numCells)`	Single call combining **request** + **ride** + **exit**
`transferTo(request, nextConveyor, entryLocation)`	Transfer to a **different** conveyor: suspend while accessing cells on the next one, then exit the current

For plan-dependent space, pair a `mapOf(testPlan to cellsNeeded)` with `numCellsNeeded = cellsNeeded` on the `convey(...)` call.

Conveyor Statistics

Statistic	Meaning
`Conveyor:loc:AccessQ:NumInQ`	Number waiting at `loc` to get cells
`Conveyor:loc:AccessQ:TimeInQ`	Time waiting at `loc` for cells
`Conveyor:from->to:NumOccupiedCells`	Occupied cells in that segment
`Conveyor:from->to:CellUtilization`	Fraction of that segment's cells occupied over time
`Conveyor:NumOccupiedCells`	Total occupied cells across the whole conveyor

Drills

This chapter's question bank has no numeric items — these are practice problems built from the formulas and rules it tests in other formats. Expect the quiz to test the same relationships, not these exact numbers.

1. In the resource-constrained test and repair model, how many workers are in the transport pool, and how many distinct workers exist in the shop?
2. A conveyor has three segments of lengths 70, 40, and 60 with `cellSize = 1`. How many cells does it have, and how are they numbered?
3. Which allocation rule should you choose to bring movers from the periphery back toward central activity, and which is the default?
4. Write, in order, the six steps that `transportWith` performs.
5. On an accumulating conveyor, an entity processes at a station while still riding. What happens to the queue length reported **at the worker**, and why?

Common Traps

- `transportWith()` does **not** require a manual `seize()` first.
- `moveWith()` requires the entity and mover to start at the **same** location.
- Conveyor segment locations are `Strings`; conveyors do **not** use a spatial model.
- Conveyor cells are numbered **from 1**, not from 0.
- `exitConveyor()` **takes simulation time** — it is not instantaneous.
- After `requestConveyor` the entity **holds cells but is not riding**.
- Seize the **worker before** the machine; release the **machine before** the worker.
- `DistancesModel` distances **may** be asymmetric, though `symmetric` defaults to true.
- The default movable-resource allocation rule is `Closest...`.

Appendix — Drill Answers

1. **Eight** workers in the transport pool, and **eight** distinct workers in the shop — every worker belongs to the transport pool as well as to its station pool.
2. $70 + 40 + 60 = \mathbf{170}$ cells, numbered **1 at the entry of the first segment through 170 at the exit of the last**.
3. `FurthestMovableResourceAllocationRule`; the default is `ClosestMovableResourceAllocationRule`.
4. seize → move empty to the entity → loading delay → `moveWith` to the destination → unloading delay → release.
5. Essentially **zero**. The conveyor keeps running and the waiting parts pile up **on the conveyor as occupied cells**, so the queue physically lives on the conveyor rather than at the worker.