Process View Modeling

Quiz Study Guide — Chapter 6

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, and multi-slot fill-in-the-blank.
- The **Common Traps** slides are built from the actual wrong answers — read them last.

The one-sentence distinction

In the **event view** the system reacts to events. In the **process view** the movement of entities through their processes **implies** the events.

What This Chapter Covers

The process-view core

- `ProcessModel`, `Entity`, `KSLProcess`
- `seize`, `delay`, `release`, `use`
- `ResourceWithQ` and `RequestQ`
- `EntityGenerator`

Coordination

- `HoldQueue`, `Signal`, `Suspension`
- `BlockingQueue`: send and `waitForItems`
- `Blockage` and `BlockingActivity`
- `waitFor` in its three overloads

Objectives

You should be able to:

- Name the class a model must extend and how `Entity` relates to `QObject`
- Say what `seize()` returns and why `release()` takes it
- List the `KSLProcessBuilder` suspending functions and what each does
- Choose the right coordination construct for a given situation, and say why
- Describe the three internal queues of a `BlockingQueue`
- Explain the constraints on a `Blockage`
- Recognize the Kotlin idioms: `::Customer`, `return@process`, qualified `this`

The Architecture

Class	Role
`ProcessModel`	Subclass of `ModelElement` containing the coroutine architecture; manages cleanup
`Entity`	**Inner class** of `ProcessModel`, subclassing `QObject`, so waiting statistics are automatic
`KSLProcess`	Interface wrapping the coroutine; exposes `isCreated`, `isSuspended`, `isRunning`, `isCompleted`, `isTerminated`, `isActivated`
`KSLProcessBuilder`	The receiver inside `process { ... }` exposing the suspending functions
`EntityGenerator`	Inner class of `ProcessModel` creating entities and activating their default process

Under the Hood

A `KSLProcess` is implemented as a **specially constructed Kotlin coroutine**, mapped onto a `ProcessCoroutine` inner class — **not** Java threads and not a `ScheduledExecutorService`.

Process state properties: `isCreated`, `isSuspended`, `isRunning`, `isCompleted`, `isTerminated`, `isActivated`, plus `processElapseTime` — completion time minus start time, valid only after completion.

Entity states in the `EntityState` hierarchy include `CreatedState`, `Scheduled`, `Active`, `WaitingForResource`, and `InHoldQueue`. There is no `CompletedState` — entities terminate via *process* states.

The Basic Process

```
val pharmacyProcess: KSLProcess = process {
    wip.increment(); timeStamp = time
    val a = seize(resource = pharmacists)
    delay(delayDuration = serviceTime)
    release(allocation = a)
    ...
}
```

- `process { ... }` returns a **`KSLProcess`**.
- `seize()` returns an **`Allocation`** — it records *which* resource and queue were involved, so `release(allocation = a)` deallocates the right resource and checks the right queue.
- The entity does **not** suspend at `seize()` if a unit is immediately available.
- `delay()` is implemented by **scheduling an event** for the end of the delay; the coroutine resumes at that point when the event fires.

The Suspending Functions

Table 1: `waitFor` is **overloaded** on three argument types — given a `KSLProcess` it activates it and suspends until it completes. Note that `simulate` is **not** a `KSLProcessBuilder` function

Function	Purpose
`seize`	Request resource units; suspend if unavailable; return an `Allocation`
`delay`	Schedule an event for the duration; suspend until it fires
`release`	Return units of a resource via the `Allocation`
`use`	Convenience: **seize, then delay, then release** in one call
`waitFor`	Suspend until a `KSLProcess`, `Signal`, or `Blockage` releases it

Resources in the Process View

- `ResourceWithQ` is a resource with a built-in `RequestQ` that holds waiting seize requests. It automatically tabulates `NumBusyUnits`, `InstantaneousUtil`, `ScheduledUtil`, `WIP`, and `SeizeCount`.
- `RequestQ` is a `Queue` subclass for seize requests. Supply it via `seize(resource, queue = requestQ)` so the entity waits in *that* queue — useful when two activities share one resource but need separate lines.
- `ResourceWithQCIfc` is a **controlled-access view**: clients can read statistics and configure non-running-state properties, but cannot mutate runtime state.
- `Allocation` pairs an entity with a resource and a queue.

The Entity Generator

```
private val generator = EntityGenerator(
    entityCreator = ::Customer,
    timeUntilTheFirstEntity = tba,
    timeBtwEvents = tba
)
```

- `::Customer` is a **function reference to the no-argument constructor** of the `Customer` inner class.
- The generator's `generate()` calls `activate(entity.defaultProcess!!)`, using a `GeneratorActionIfc` inherited from `EventGenerator`.

Default Processes

- Designate the default with **`isDefaultProcess = true`** on the `process()` builder, or by setting the entity's `defaultProcess` property directly.
- With **no** default process, the generator throws an **`IllegalStateException`**.
- A `process()` given a **`name`** is added to the entity's **`processes` map**, keyed by that name, where it can be retrieved later.
- `activate(c.pharmacyProcess)` **schedules the start** of the coroutine — typically at the current time with a default priority — so it begins after the current event finishes. It does **not** run the process synchronously.

Trap

An entity experiences **exactly one process at a time**. Activating a second process for an entity already running one is an error.

Coordination Constructs

Construct	Description
`HoldQueue`	A `Queue` subclass; entities **cannot self-remove** — other code calls `removeAndResume(entity)` or `removeAllAndResume()`. Collects queueing statistics.
`Signal`	Built on `HoldQueue`; `signal(range)` notifies entities **by rank** and they self-remove
`BlockingQueue`	`senderQ`, receiver `RequestQ`, and `channelQ`; suspends senders and receivers
`Suspension`	Low-level `suspendFor()` / `resume()` pair; **no queueing statistics**
`Blockage`	Semaphore-like gate over suspending code: `startBlockage(b)` / `clearBlockage(b)`

Choosing a Coordination Construct

- `HoldQueue` and `Signal` are built on `Queue`, so they **automatically collect waiting statistics**. Raw `Suspension` does not.
- So: **when waiting statistics are required, use `HoldQueue` or `Signal`, never `Suspension`.**
- An entity in a `HoldQueue` **cannot remove itself** — other code must call `removeAndResume(entity)` or `removeAllAndResume()`.
- A `Signal` is layered on a `HoldQueue`, and signaled entities **do** self-remove.
- In `SignalExample`, `signal(0..4)` is a Kotlin range giving the **ranks** (positions 0 through 4) of the entities in the hold queue to notify.

The Blocking Queue

Three internal queues:

- **`senderQ`** — entities blocked at `send()` because the channel is full
- **receiver `RequestQ`** — entities blocked at `waitForItems()`
- **`channelQ`** — the actual channel holding items

Default capacity is **`Int.MAX_VALUE`**, so senders effectively never block on a full channel unless a capacity is supplied. On a truly full channel, `send()` **suspends** the sender in `senderQ`.

Selection: the default rule fills **the next request in the request queue**.
`FirstFillableRequest` is a `RequestSelectorIfc` that scans the waiting queue and returns the **first request that** `canBeFilled`, or null.

Tie-Dye T-Shirts

```
completedShirts = waitForItems(completedShirtQ, size, { it.orderNum == this@Order
```

- The lambda is a **predicate** used by the blocking queue to test which items match this order's id, so an order consumes **only its own shirts**.
- Each shirt's process ends with `send(this@Shirt, completedShirtQ)` — issued by the **Shirt** entity, not the Order.
- The Order activates shirts with `activate(shirt.shirtMaking)` in a loop. Those activations are scheduled at the **current simulation time**, but because the current event is still executing they remain pending until subsequent events fire — this is the source of **pseudo-parallelism**.
- The order uses `seize(myPackager, queue = myOrderQ)` for paperwork and `seize(myPackager)` for final packaging: **same resource, different waiting queues**.

Blockages

- Started with `startBlockage(b)` and cleared with `clearBlockage(b)`; entities calling `waitFor(b)` suspend while the section is started.
- **Only the entity that instantiates the blockage may start and clear it.**
- A started blockage **must** be cleared before the process ends, or an exception is thrown. It is **not** silently cleared for you.
- `BlockingActivity` specializes `Blockage` to wrap a delay; use it with the `perform(blockingActivity)` suspending function so the blockage is always cleared.
- `BlockingResourceUsage`, `BlockingResourcePoolUsage`, and `BlockingMovement` wrap blockages around `use()`, pool `use()`, and `move()` so you need not manage start/clear by hand.

Kotlin Idioms Worth Knowing

- `return@process` — a labeled return targeting the `process` builder; used to exit early, as for a "leaver" student in the STEM mixer.
- **Qualified `this`** (`this@Order`, `this@Shirt`, `this@Customer`) — disambiguates which enclosing entity instance is meant, since the builder lambda has its own implicit receiver, the `KSLProcessBuilder`.
- `toBoolean()` — an extension function in `KSLRandom` converting 1.0 to true and 0.0 to false, as in `myDecideToWander.value.toBoolean()`.
- In the STEM mixer the student **type** is an entity **attribute**, not a separate `Entity` subclass per type.

Structure and Cleanup

Recommended layout of a `ProcessModel` subclass:

1. Subclass `ProcessModel`
2. Declare random variables
3. Declare responses
4. Declare resources, queues, generators
5. Declare entity inner classes with their `process()` definitions

At the end of a replication, `ProcessModel` **terminates entities still suspended** and ensures no entity terminates while still holding resource allocations.

Because both views describe the same underlying stochastic system, the process-view pharmacy model reproduces the Chapter 4 event-view results **exactly**, given the same streams and run parameters.

Common Traps

- An entity experiences **one** process at a time — never several in parallel.
- `seize()` returns an `Allocation`, and the entity **does not** suspend when a unit is immediately available.
- A started `Blockage` **must** be cleared, and only its instantiating entity may start or clear it.
- An unbounded `BlockingQueue` means senders never block on a full channel — but a *bounded* one **suspends** the sender rather than throwing or discarding.
- The **Shirt** sends itself to the channel, not the Order.
- `HoldQueue` collects statistics; raw `Suspension` does **not**.
- STEM mixer student types are attributes, not subclasses.
- `simulate` is not one of the `KSLProcessBuilder` suspending functions.

Quick Self-Test

Without looking back, answer these:

1. What does `seize()` return, and why does `release()` take that rather than the resource?
2. Which two coordination constructs collect queueing statistics, and which one does not?
3. Name the three internal queues of a `BlockingQueue` and what suspends in each.
4. What are the two constraints on a `Blockage`?
5. What happens if an `EntityGenerator` is given an entity type with no default process?
6. What are the three things `waitFor` can be given?