Post-Task 3

- Reflect on Task 3
  - what was easiest to understand in the requirements/specifications?
  - what was hardest to understand in the requirements/specifications?
  - what was easiest to approach solving?
  - what was hardest to approach solving?
  - what was easiest to solve?
  - what was hardest to solve?
  - what was your testing strategy?
  - how early did you start?
  - for whatever did not work or you did not do, explain why
  - if you had to do this task over again, how would you improve your process?
  - how would you explain the quality of your work to your boss/teammates ?
  - in which year and quarter did you take CS 210 and 211?
  - who was/were your instructor(s)
- 350 words, due Friday
  - indicate word count

CS 350 Task 3: Proof of Concept Feasibility Grounding [UPDATED]

Description

This task plays an experimental role for implementing the expected primary data, control, and behavior of one of the two critical components in the project. It serves as the underlying computational mechanism to drive a hydraulic cylinder in a semi-realistic manner (consistent with so-called "naive physics"). For logistical reasons, the only motion we support is linear.

The second critical component is a three-dimensional box with arbitrary dimensions, orientation, and position, which can be connected to other such boxes via static and dynamic linkages. This task addresses the latter. The former and the box itself play no role here. Task 4 (which was originally intended to come before this one) provides the basis for deciding that these two components are the critical ones to experiment with.

The task consists of three parts:

Pretask:	A free-form request to me for clarification of this work.
Actual Task:	Defined below.
Posttask:	A reflection on your solution and its development process after receiving graded feedback on it. The format and deadline of this part will be provided later.

Requirements

You are writing a simple Java framework for creating actuators of two related types. Both are a form of counter that maintains an arbitrary double value as a state on an interval. The state is initialized to the start of the interval and advances toward the end by a step once per update. The linear actuator advances by a fixed step. The nonlinear actuator advances by a step that itself changes once per update, which models simplistic acceleration.

An appropriate solution requires only programming skills that you already have from the previous courses.

Specifications

I. General

a. All code must reside in package `w15cs350task3`.
b. Include appropriate `toString()` methods.
c. Include appropriate error checking that throws a `RuntimeException`.
d. Unless otherwise specified, you may implement your solution any way you want.
e. You must document your code with basic Javadoc.
f. Use proper object-oriented programming, and especially consider the extensibility of your solution.

II. Create an interface called `I_Actuator` with the following contract:

`String getID_()`
Gets the arbitrary nonempty identifier of this component.
`double getState_()`
Gets the current state, which is always between the start and end states.
`double getStateStart_()`
Gets the initial state that this state assumes.
`double getStateEnd_()`
Gets the final state that this state can assume.
`double getStep_()`
Gets the value of the transition step between the start and end states. It is always positive.
`boolean updateState_()`
Updates the state from its current state to its next state based on the current step. If the next state exceeds the end state, then the former is clamped to the latter. This returns whether the end state has been reached.

void cancel_()
Immediately stops the component from servicing calls to `updateState_()`.

void terminate_()
Stops the component from servicing calls to `updateState_()` with a notional gradual shutdown. For linear components, reverse the step and cease servicing three calls after receipt of this terminate signal. For nonlinear components, also reduce the step by half each time.

boolean isDying_()
Returns whether the component is being terminated.

boolean isDead_()
Returns whether the component has been terminated or canceled.

III. Create a concrete public class called `MyActuatorLinear` that implements `I_Actuator` such that calling `updateState_()` increments or decrements the current state by the step once per call until the end state is reached.

The constructor is:

```
MyActuatorLinear(String id, double stateStart, double stateEnd, double step)
```

IV. Create a concrete public class called `MyActuatorNonlinear` that implements `I_Actuator` such that calling `updateState_()` increments or decrements the current state by the current step once per call until the end state is reached, and the step value then increments by **`stepAcceleration`**.

The constructor is:

```
MyActuatorNonlinear(String id, double stateStart, double stateEnd, double step,
                    double stepAcceleration)
```