Example: Model Train Controller

Purposes of example:
Follow a design through several levels of abstraction.
Gain experience with UML.

Text: Section 1.4
Model train setup

console
power supply
rcvr
motor

classical
header address command ECC
Requirements

• Console controls up to 8 trains on 1 track.
• Throttle has at least 63 levels.
• Inertia control adjusts responsiveness with at least 8 levels.
• Emergency stop button.
• Error detection scheme on messages.
  • Ignore erroneous messages
# Requirements form

<table>
<thead>
<tr>
<th>Name</th>
<th>Model train controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Control speed of &lt;= 8 model trains</td>
</tr>
<tr>
<td>Inputs</td>
<td>Throttle, inertia, emergency stop, train #</td>
</tr>
<tr>
<td>Outputs</td>
<td>Train control signals</td>
</tr>
<tr>
<td>Functions</td>
<td>Set engine speed w. inertia; emergency stop</td>
</tr>
<tr>
<td>Performance</td>
<td>Can update train speed at least 10 times/sec</td>
</tr>
<tr>
<td>Cost</td>
<td>$50</td>
</tr>
<tr>
<td>Power</td>
<td>Wall powered</td>
</tr>
<tr>
<td>Physical</td>
<td>Console comfortable for 2 hands; &lt; 2 lbs.</td>
</tr>
</tbody>
</table>
Before we create a detailed specification, we will make an initial, simplified specification.

* Gives us practice in specification and UML.
* Good idea in general to identify potential problems before investing too much effort in detail.
## Basic system commands

<table>
<thead>
<tr>
<th>Command-name</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-speed</td>
<td>speed (positive/negative)</td>
</tr>
<tr>
<td>set-inertia</td>
<td>inertia-value (non-negative)</td>
</tr>
<tr>
<td>estop</td>
<td>none</td>
</tr>
</tbody>
</table>
Typical control sequence

Console always monitoring buttons/knobs

Time

Receiver always "listening"

:console

- set-inertia
- set-speed
- set-speed

:train_rcvr

- estop
- set-speed

Set-speed

Set-speed

Set-speed
Implemented message classes derived from message class.

- Attributes and operations will be filled in for detailed specification.

- Implemented message classes specify message type by their class.

- May have to add type as parameter to data structure in implementation.
Subsystem collaboration diagram

Shows relationship between console and receiver (ignores role of track): interaction via commands
System structure modeling

• Some classes define non-computer components.
  • Denote by *name.
• Choose important systems at this point to show basic roles and relationships.

Major subsystem roles

• Console:
  • read state of front panel;
  • format messages;
  • transmit messages.

• Train:
  • receive message;
  • interpret message;
  • control the train.
- **panel**: describes analog knobs and interface hardware.
- **formatter**: turns knob settings into bit streams.
- **transmitter**: sends data on track.

* = physical object
Train system class diagram

- **receiver**: digitizes signal from track.
- **controller**: interprets received commands and makes control decisions.
- **motor interface**: generates signals required by motor.
Detailed specification

- We can now fill in the details of the conceptual specification:
  - more classes;
  - behaviors.
- Sketching out the spec first helps us understand the basic relationships in the system.
Train system analog physical object classes

**Knobs**
n
- train-knob: integer
- speed-knob: integer
- inertia-knob: unsigned-integer
- emergency-stop: boolean

**Pulser**
n
- pulse-width: unsigned-integer
- direction: boolean

**Sender**

- send-bit()

**Detector**

- read-bit() : integer

Motor controlled by pulse width modulation:

V

Duty cycle

+ -- | -- -
Panel and motor interface classes

- **panel** class defines the controls.
  - new-settings() function reads the controls.
- **motor-interface** class defines the motor speed/inertia, held as state.

<table>
<thead>
<tr>
<th>panel</th>
<th>motor-interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>train-number() : integer</td>
<td>speed: integer</td>
</tr>
<tr>
<td>speed() : integer</td>
<td>inertia: integer</td>
</tr>
<tr>
<td>inertia() : integer</td>
<td></td>
</tr>
<tr>
<td>estop() : boolean</td>
<td></td>
</tr>
<tr>
<td>new-settings()</td>
<td></td>
</tr>
</tbody>
</table>
Control input cases

• Use a soft panel to show current panel settings for each train.

• Changing train number:
  • must change soft panel settings to reflect current train’s speed, etc.

• Controlling throttle/inertia/estop:
  • read panel, check for changes, perform command.
Transmitter and receiver classes

- **Transmitter class** has one method for each type of message sent.
- **Receiver class** provides methods to:
  - detect a new message;
  - determine its type;
  - read its parameters (estop has no parameters).

<table>
<thead>
<tr>
<th>Transmitter</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-speed(adrs: integer,</td>
<td>current: command</td>
</tr>
<tr>
<td>speed: integer)</td>
<td>new: boolean</td>
</tr>
<tr>
<td>send-inertia(adrs: integer,</td>
<td>read-cmd()</td>
</tr>
<tr>
<td>val: integer)</td>
<td>new-cmd() : boolean</td>
</tr>
<tr>
<td>send-estop(adrs: integer)</td>
<td>rcv-type(msg-type: command)</td>
</tr>
<tr>
<td></td>
<td>rcv-speed(val: integer)</td>
</tr>
<tr>
<td></td>
<td>rcv-inertia(val: integer)</td>
</tr>
</tbody>
</table>
### Formatter class

- **Formatter class** holds state for each train, setting for current train.
- The **operate()** operation performs the basic formatting task.

<table>
<thead>
<tr>
<th>formatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>current-train: integer</td>
</tr>
<tr>
<td>current-speed[ntrains]: integer</td>
</tr>
<tr>
<td>current-inertia[ntrains]: unsigned-integer</td>
</tr>
<tr>
<td>current-estop[ntrains]: boolean</td>
</tr>
<tr>
<td>send-command()</td>
</tr>
<tr>
<td>panel-active() : boolean</td>
</tr>
<tr>
<td>operate()</td>
</tr>
</tbody>
</table>
Control input sequence diagram

:knobs*

change in speed/inertia/estop

change in control settings

change in train number

set-knobs

:panel

read panel

panel settings

read panel

panel settings

new-settings

:formatter

panel-active

send-command

send-speed, send-inertia, send-estop

:transmitter

send-command
Formatter *operate()* behavior
(in the formatter class)

Diagram:
- **idle**
- **panel-active()**
- **update-panel()**
- **new train number**
- **send-command()**
- **other**

arrow connections:
- From **idle** to **panel-active()**
- From **panel-active()** to **update-panel()**
- From **panel-active()** to **new train number**
- From **panel-active()** to **send-command()**
- From **panel-active()** to **other**
- From **other** to **idle**
Formatter **panel-active()** behavior
(in the formatter class)

```
panel*:read-train()
```

- current-train != train-knob

```
panel*:read-speed()
```

- current-speed != throttle

```
current-train = train-knob
update-screen
changed = true
```

```
current-speed = throttle
changed = true
```

...
Train controller class

controller

- current-train: integer
- current-speed[ntrains]: integer
- current-direction[ntrains]: boolean
- current-inertia[ntrains]: unsigned-integer

operate()
issue-command()
Setting the speed

- Don’t want to change speed instantaneously.
- Controller should change speed gradually by sending several commands.
Controller operate behavior

wait for a command from receiver

receive-command() --> issue-command()
Sequence diagram for set-speed cmd.

:detector*

:receiver

new-cmd

cmd-type

rcv-speed

:controller

set-speed

:motor-interface

set-pulse

: pulser*

set-pulse

set-pulse

set-pulse

set-pulse
Refined command classes

- **command**
  - type: 3-bits
  - address: 3-bits
  - parity: 1-bit

- **set-speed**
  - type=010
  - value: 7-bits

- **set-inertia**
  - type=001
  - value: 3-bits

- **estop**
  - type=000
Summary

- Separate specification and programming.
  - Small mistakes are easier to fix in the spec.
  - Big mistakes in programming cost a lot of time.
- You can’t completely separate specification and architecture.
  - Make a few tasteful assumptions.