Formalisms for System Design

Dongbing Gu

School of Computer Science and Electronic Engineering
University of Essex
UK

Spring 2018

Outline

Section 1

Embedded System Design Process

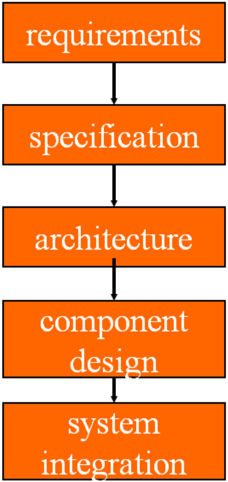
Design Methodologies

- Process for creating a complex systems.
  - everyone has a design process in mind when designing an embedded system;
  - multiple designer team demands a design process.
- Many systems are complex:
  - large specifications;
  - multiple designers;
  - interface to manufacturing.
- Proper design processes improve:
  - quality;
  - cost of design and manufacture.

Design Goals

- Functionality and user interface.
- Performance - overall speed, etc.
- Manufacturing cost.
- Power consumption.
- Design cost for a few copies is different with mass market.
- Time-to-market:
  - beat competitors to market;
  - meet marketing window.
- Other requirements (physical size, etc.)

Levels of Abstraction



Top-down vs. Bottom-up

- Top-down design:
  - start from most abstract description; work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.
- May be partially or fully automated, such as using software tools to transform, verify design.

Requirements

- Requirements: informal description of what customer wants using plain language.
- Specification: precise description of what design team should deliver.
- Requirements phase links customers with designers.

Types of requirements

- Functional: input/output relationships.
- Non-functional:
  - timing: time required to compute output;
  - power consumption;
  - manufacturing cost;
  - physical size, weight, etc.;
  - time-to-market;
  - reliability.

Creating Requirements

- Customer interviews.
- Comparison with competitors.
- Sales feedback, talking to marketing representatives;
- Prototypes, providing prototypes to users for comment;
- Next-bench syndrome: the engineers are most comfortable designing products for their colleagues sitting next to them.

Requirements

- Five different influences that can generate requirements during the design of an embedded system:
  - Stakeholders (end-users, customers, managers, engineers, maintenance experts, and certification bodies).
  - Technical constraints
  - Industry standards
  - Quality assurance
  - Sales and Marketing

Good requirements

- Correct
- Unambiguous
- Complete (all requirements should be included)
- Consistent: requirements do not contradict each other.
- Verifiable: is each requirement satisfied in the final system?
- Modifiable: shall be structured, and can update requirements easily.
- Traceable:
  - know why each requirement exists;
  - go from source documents to requirements;
  - go from requirement to implementation;
  - back from implementation to requirement.

Requirements Form

Name	Assignment 1
Purpose	
Inputs	
Outputs	
Functions	
Performance	
Manufacture costs	
Physical size/weight	
Power	

Specifications

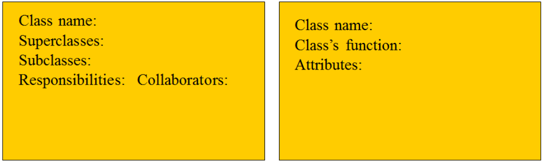
- A more precise description of the system, should provide input to the architecture design process.
- Capture functional and non-functional properties, need to verify the correctness
- Many specification styles:
  - control-oriented vs. data-oriented;
  - textual vs. graphical.
- May be executable or may be in mathematical form for proofs.
- UML is one specification and design language

Architecture Design

- What major components satisfy the specification?
- Hardware components: CPUs, peripherals, etc.
- Software components: major programs and their operations.
- Must take into account functional and non-functional specifications.

Architecture Design - CRC cards

- Well-known method for analysing a system and developing an architecture.
- CRC stands for the following three major items:
  - Classes define the logical groupings of data and functionality.
  - Responsibilities describe what the classes do.
  - Collaborators are other classes with which a given class works.
- CRC is a team-oriented methodology:
- It is used to turn specification into architecture design.



front back

CRC cards

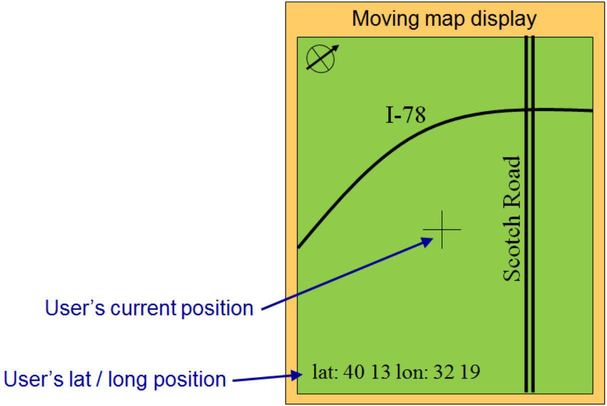
- They are physical cards held by members of the team.
- Group members write these cards, talk about them, and update the cards until they are satisfied with the results.
- All team members understand all parts of the system and how they interact, and to reveal any deficiencies in the current design.

Component Design and System Integration

- Designing hardware and software components.
  - Must spend time designing the system before you start coding.
  - Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.
- System integration
  - Put together the components. Many bugs appear only at this stage.
  - Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

Example: GPS Moving Map

- Moving map obtains position (Latitude and Longitude) from GPS, paints map from local database.



Example: GPS Moving Map Requirements

- Functionality:
  - For automotive use. Show roads and other landmarks available.
- User interface:
  - At least 400×600 pixel screen.
  - Three buttons, a menu should pop-up when buttons pressed to allow user's selection.
- Performance:
  - Map should scroll smoothly.
  - No more than 1 sec power-up. verify and display GPS information within 15 seconds.
- Cost: less than £100 shop price - about £40 cost of hardware.
- Physical size/weight: Should fit in dashboard.
- Power consumption: Current draw comparable to CD player.

GPS Moving Map Requirements Form

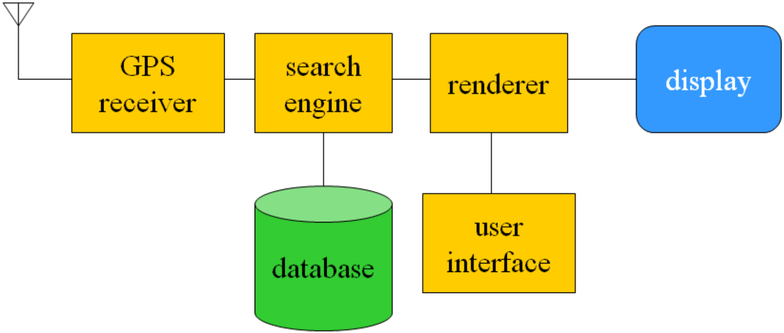
Name	GPS moving map
Purpose	consumer-grade moving map for driving
Inputs	power button, two control buttons
Outputs	back-lit LCD 400×600
Functions	receiver; 3 user selectable resolutions; displays current lat/long
Performance	updates screen within 0.25 sec of movement
Manufacture costs	£40
Physical size/weight	no more than 2×6 inches, 12 oz
Power	100mW

GPS Specification

- what is received from GPS;
- map data;
- user interface;
- operations required to satisfy user requests;
- background operations needed to keep the system running.

GPS Moving Map Block Diagram

- Block diagram: major operations and data flows among them.



Section 2

Visual Modelling Language UML

System Modelling - UML

- Modelling is an essential part of embedded system design.
- Models help at a higher level of abstraction by hiding or masking details, bringing out the big picture, or by focusing on different aspects of the prototype.
- Unified Modelling Language (UML) is a standard visual modelling language.
- Built on fundamental Object-Oriented concepts including class and operation, it's a natural fit for object-oriented languages and environments.
- UML is typically used as a part of software development process. It can be also used for embedded system development process.

Object-Oriented (OO) Design

- It encourages the design to be described as a number of interacting objects.
- At least some of those objects will correspond to real pieces of software or hardware in the systems.
- Some objects will closely correspond to real-world objects. Some objects may be useful only for description or implementation.
- Objects provide interfaces to read/write state, hiding the object's implementation from the rest of the system.

UML 2.0

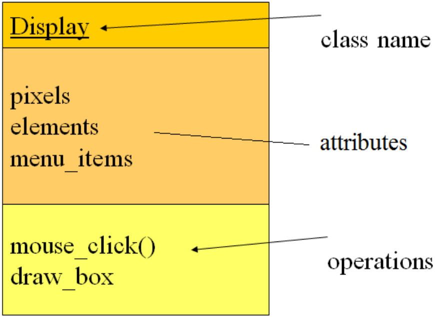
- UML 2.0 defines thirteen types of diagrams, divided into three categories: Six diagram types represent static structure; three represent general types of behaviour; and four represent different aspects of interactions:
  - Structure Diagrams include the **Class Diagram, Object Diagram**, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
  - Behaviour Diagrams include the Use Case Diagram; Activity Diagram, and **State Machine Diagram**.
  - Interaction Diagrams, all derived from the more general Behaviour Diagram, include the **Sequence Diagram**, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

Section 3

Structure diagrams

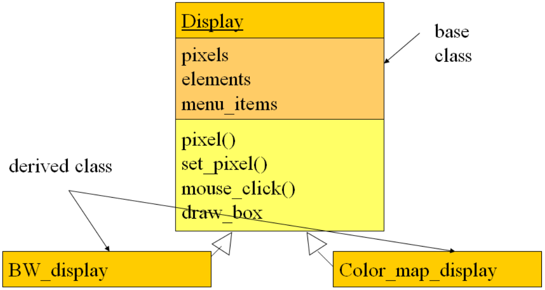
Structure diagram - Class diagram

- A class is a classifier which describes a set of objects that share the same features, constraints, semantics (meaning).
- Features of a class are attributes and operations.
- Class diagram shows structure of the designed system at the level of classes, and shows their features, constraints and relationships.



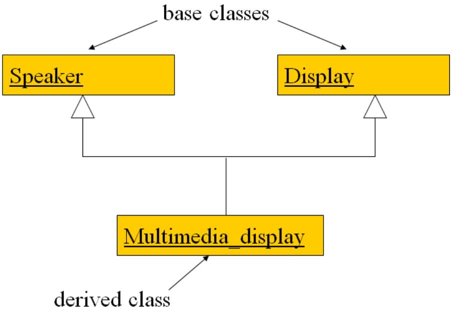
Class Diagram - Generalisation

- UML allows to define one class from another.
- A derived class inherits all the attributes and operations from its base class.
- UML considers inheritance to be one form of generalisation, shown with a hollow triangle as an arrowhead.
- Example: BW_display and Colour_map_display are specific versions of Display. Display generalises both of them.



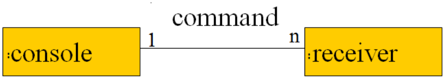
Multiple inheritance

- UML allows to define multiple inheritance, in which one class from more than one base class.

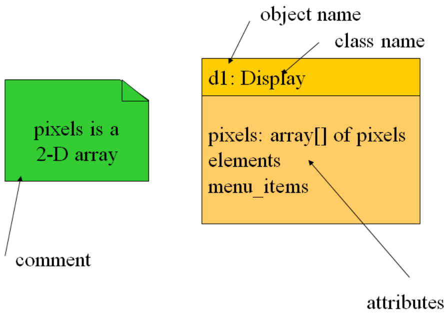


Class diagram - Association

- Association is a relationship between classes.
- It is drawn as a solid line connecting two classes. Name of the association can be shown somewhere near the middle of the association line.
- A number at the ends shows the multiplicity.



- Object diagram is instance level class diagram which shows instance specifications of classes.
- It shows a snapshot of the detailed state of a system at a point in time.
- Objects of a class must contain values for each attribute.
- All objects derived from the same class have the same attributes, but may have different attribute values.

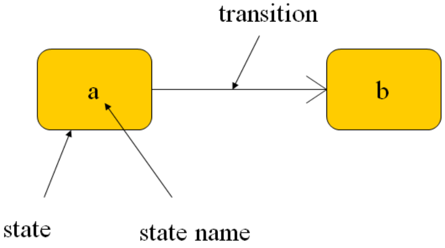


Section 4

Behaviour Diagrams

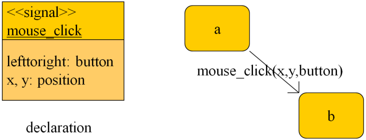
Behaviour Diagrams - State Machine Diagram

- Behaviour diagrams show the dynamic behaviour of the system, which can be described as a series of changes to the system over time.
- State machine diagram shows discrete behaviour of a part of designed system through finite state transitions.
- Behaviour is modelled as a graph of state nodes connected with transitions.
- Transitions are triggered by the occurrence of events.
- An event may come from inside or outside of the system.

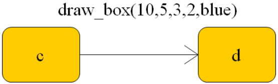


Types of events

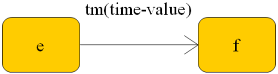
- Signal: asynchronous event, defined by an object in UML.



- Call event: synchronised communication, such as a procedure call in a programming language.

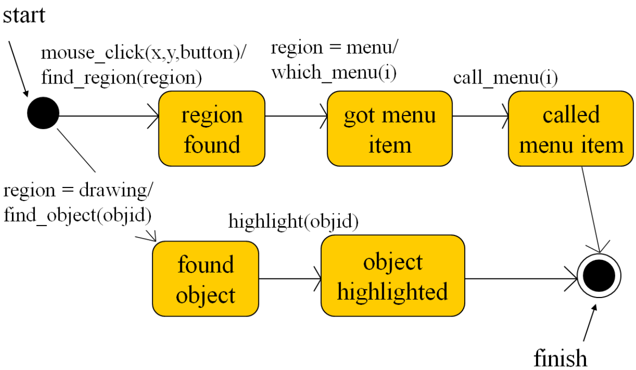


- Time-out event: causes the machine to leave a state after a certain amount of time.



Example State Machine Diagram

- State models a situation during which some conditions hold, such as waiting for some external event to occur.
- State is shown as a rectangle with rounded corners and the state name inside the rectangle.
- An initial state is shown as a small solid filled circle.
- A final state is shown as a circle surrounding a small solid filled circle.

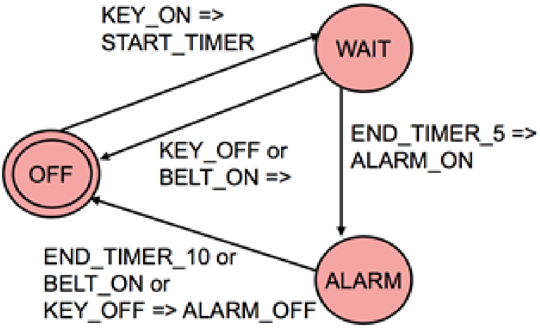


Example State Machine Diagram

- Draw a finite state machine diagram for an embedded system that controls the car seat belt. If the driver turns on the key, and does not fasten the seat belt within 5 seconds, then an alarm beeps for 10 seconds, or until the driver fastens the seat belt, or until the driver turns off the key.

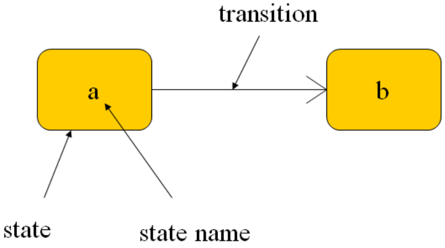
Example State Machine Diagram

- Draw a finite state machine diagram for an embedded system that controls the car seat belt. If the driver turns on the key, and does not fasten the seat belt within 5 seconds, then an alarm beeps for 10 seconds, or until the driver fastens the seat belt, or until the driver turns off the key.



State Machine Diagram Limitations

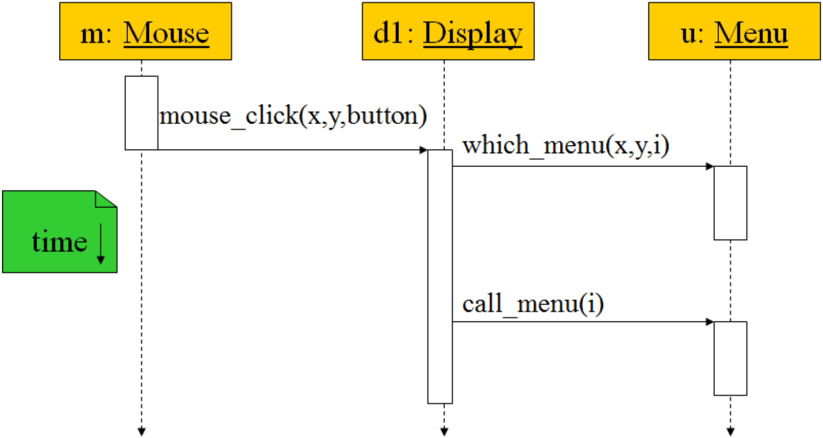
- Scalability - Number of states and transitions increase exponentially as the system complexity increases
- No concurrency support



Sequence Diagram

- Sequence diagram focuses on the message interchange between a number of lifelines.
- Lifeline represents an individual participant in the interaction.
- A lifeline is shown using a symbol that consists of a rectangle forming its "head" followed by a vertical line that represents the lifetime of the participant.
- Usually the head is a rectangle containing name of class and object.
- Execution represents a period in the participant's lifetime when it is executing a unit of behaviour or action within the lifeline.
- Execution is represented as a thin rectangle on the lifeline.
- Message occurrence represents such events as sending and receiving of signals or invoking and receiving of operation calls.

Sequence Diagram

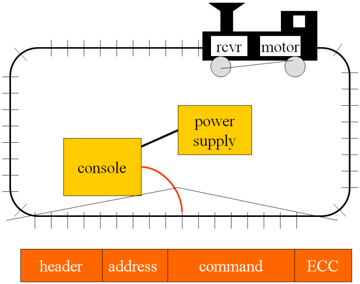


Section 5

Example: Model Train Controller

Model Train Setup

- Using this example to follow a design through several levels of abstraction, and gain experience with UML.
- The user sends messages to train with a control box attached to tracks.
- The message is modulated on the power supply voltage. The train is powered by the two rails of the track.
- The train senses the message and control the speed and direction.



Requirements

- The console can control 8 trains on 1 track.
- The speed should be controllable by a throttle to at least 63 levels in each direction.
- Inertia control allows the user to adjust responsiveness of the train to commanded changes in speed with at least 8 levels.
- Emergency stop button.
- Error detection scheme on messages.

Requirements form

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting; emergency stop; train number
Outputs	Train control signals
Functions	Set engine speed based on inertia setting; emergency stop
Performance	Update train speed at least 10 times per sec
Manufacture cost	£50
Physical size/weight	Console comfortable for 2 hands; less than 2 pounds
Power	10W (plugs into wall)

Conceptual specification

- Before creating a detailed specification, an initial and simplified specification allows us to understand the system a litter better.

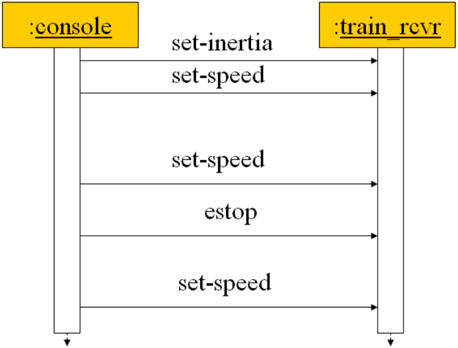
Basic System Commands

- The message determines what the controller can do.
- Defining the message first will help us understand the functionality of the components.

command name	parameters
set-speed	speed (positive/negative)
set-inertia	inertia value (nonnegative)
Estop	none

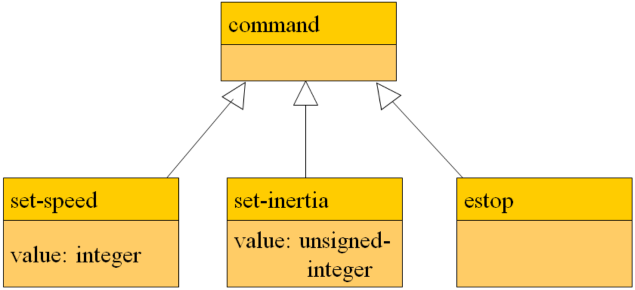
Typical Control Sequence

- Then consider how the console controls the train by sending the message over the track.
- The console can send the message at any time.
- Set-inertia message is less frequently than set-speed message.



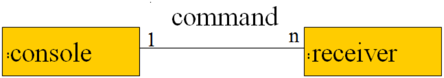
Message Classes

- The message class can be modelled into two level class hierarchy.
- One is the base class: Command.
- Three subclasses derived from Command.
- Attributes and operations will be filled in for detailed specification.



Subsystems

- There are two major subsystems: console and receiver.
- The basic relationship is shown in UML.



Console:

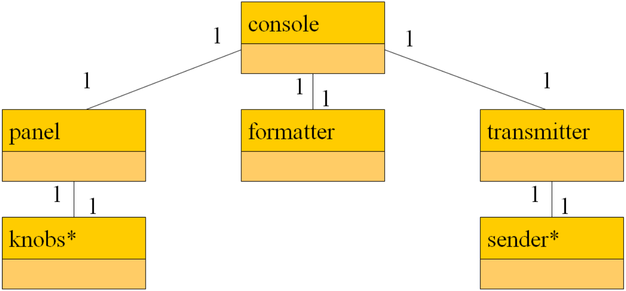
- read state of front panel;
- format messages;
- transmit messages.

Train Receiver:

- receive message;
- interpret message;
- control the train.

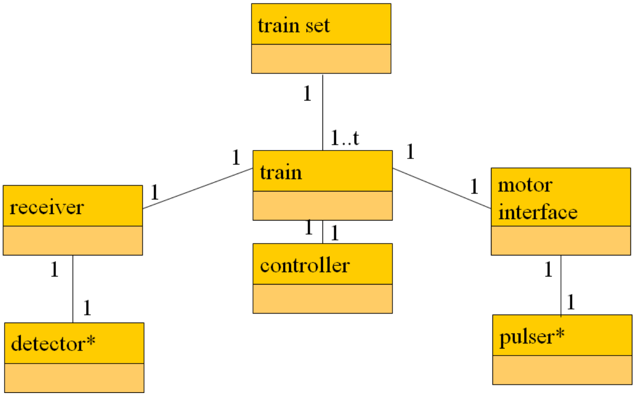
Console System Classes

- Panel class: describes the console's front panel, including analogue knobs and interface hardware.
- Formatter class: includes operations that know how to read the panel knobs and creates a bit stream.
- Transmitter class: interfaces the analogue electronics to send data on track.
- Numeric values show the number of instances of the classes.



Receiver System Classes

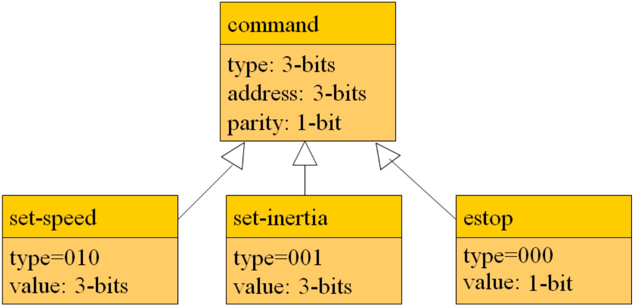
- Receiver class: reads digital signals from track.
- Controller class: interprets received commands and makes control decisions.
- Motor interface class: generates signals required by motor.



Detailed Specification

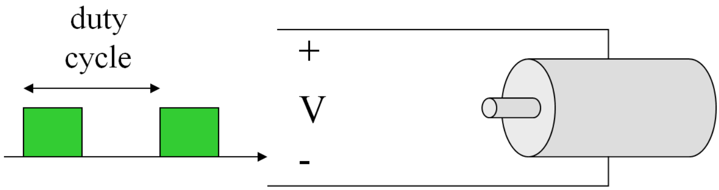
- We can now fill in the details of the conceptual specification with attributes and operations.

Refined Command Classes



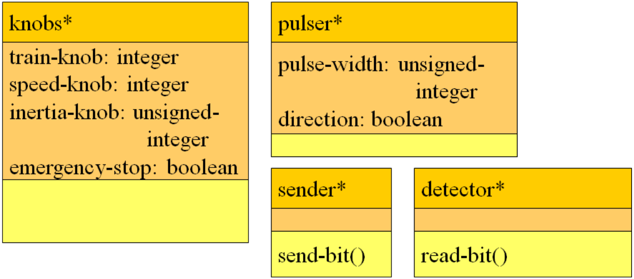
Train speed control

- Motor controlled by Pulse Width Modulation:



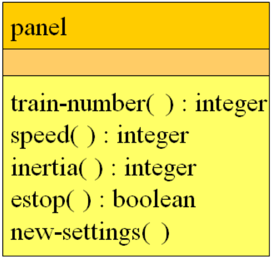
Physical Object Classes

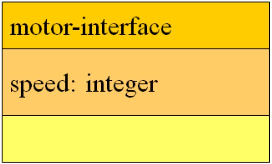
- The Panel has three knobs: train number, speed, and inertia, and one button: emergency stop.
- The Knob class specifies each of them and provides a set-knob operation that allows the rest of the system to modify the knob setting.
- The Sender and the Detector classes simply put out and pick up a bit.
- The Pulser class defines an integer to specify the speed and a separate binary for motor direction.



Panel and Motor Interface Classes

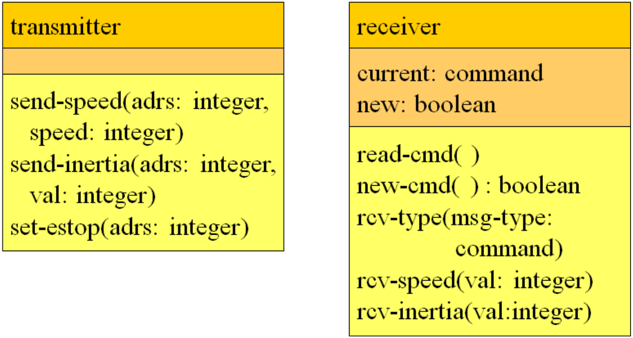
- The Panel class defines a behaviour for each of the controls on the panel. (new-settings( ) uses the set-knob operation of the Knob class to change the knob setting.)
- The Motor-interface class defines the motor speed.





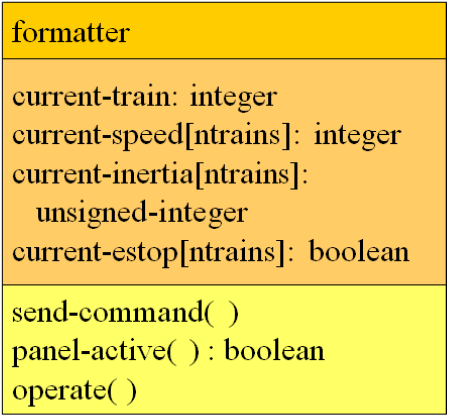
Transmitter and Receiver Classes

- The Transmitter class has one operation for each type of message sent.
- The Receiver class provides methods to:
  - detect a new message;
  - determine its type;
  - read its parameters



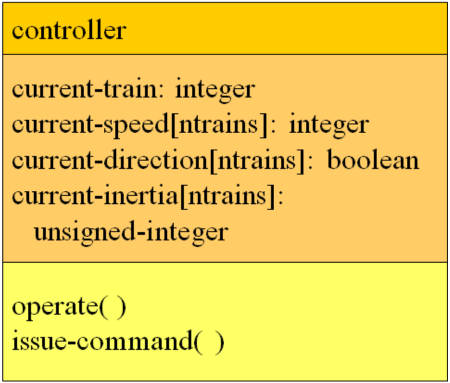
Formatter Class

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



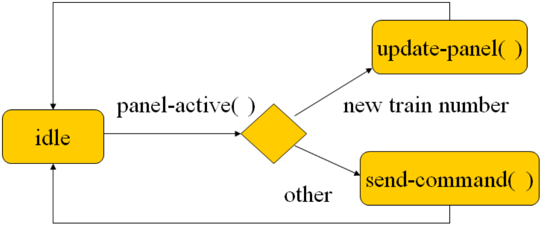
Controller Class

- The Controller class has operate() which is called by the Receiver when it gets a new command, and issue-command() which changes the speed, inertia settings.



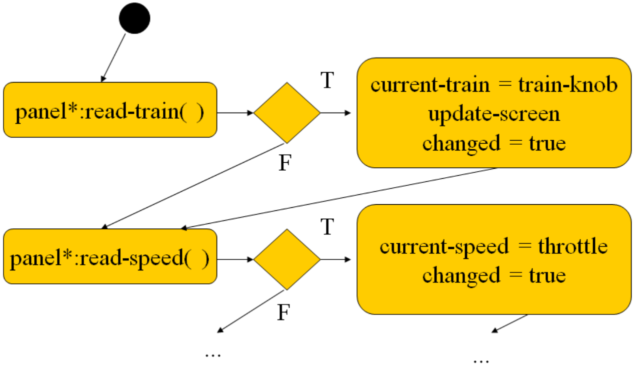
Formatter Operate() Behaviour

- State machine for a very simple version of operate().
- This operation checks the panel. If a train number changes, it updates the panel display, otherwise it sends the required message.



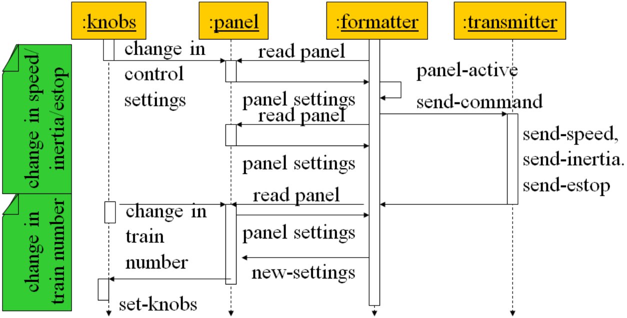
Formatter Panel-Active() Behaviour

- State machine for the panel-active operation.



Sequence Diagram for Control Input

- The Formatter periodically calls the Panel.
- Once a change is detected, a send-command is sent to the Transmitter.
- If a train number is changed, the Formatter must cause the knob setting to be reset a proper value.



Sequence Diagram for set-speed command

- The Controller operate() must determine the nature of the message
- Once the speed command has been parsed, it must send a sequence of commands to the motors to smoothly change the speed.