How is Software Different from Other Engineering?

- No natural (internal) boundaries.
  - Requires special skills to choose designs.
  - Components can interact in many ways.
- It's not constructed from materials that obey physical laws.
  - Interpolation is rarely valid.
  - Difficult to build in safety margins (can't extrapolate).
  - Doesn't wear out or break.
- Possible interactions with other systems (e.g., operating system, other programs etc.) are essentially infinite.

Steps in Software Engineering

- Plan the project
- Design the system
  - Start at 'high' level (block diagram)
  - Decompose into components (modules/classes)
- Implement
- Verify and validate.

Choosing Modules

- *Abstraction* — The interface to a module should be much simpler than the implementation.
- *Encapsulation* (a.k.a. *information hiding*) — Each module should 'hide' a design decision, such as
  - Implementation of algorithms,
  - Interface with some hardware/other system,
  - Data representation.
- Strive for simple and small interfaces between modules (low coupling).
- Normally a module should be clearly associated with a single functional block in the block diagram (high cohesion).
- The decisions to hide are the ones that are most likely to change.

Example Modules

Sensor module: Hides the interface to all sensors.

Inputs HW signals from all sensors.
Outputs Sensor state vector (8 bits) to be used by other modules.

Motor module: Implements control of all motors.

Inputs Motor commands (forward/backward/stop for each motor).
Outputs HW signals to control motors.

What's wrong with this (partial) modularization?

Better Modules

Tape detection module: Detects presence of the reflective tape.

Inputs HW signals from tape detection sensors.
Outputs Status relative to tape (e.g., on tape, off left, off right, off front, off all)

Steering module: Controls movement of vehicle.

Inputs Direction command: forward, adj left, adj right, hard left, hard right, reverse, stop.
Outputs HW signals to drive motors.

Software Design Documentation

Essential components:

- Interface — how to use it
- Behaviour — what does it do
- Relationships (association) — how does it fit within the system

Other documents:

- Test plan & results.

Example Module Design

The steering module is used by all other modules to control the direction of the motor. If the "forward" command is given, both wheel directions (pin 15 and 18) are set to logic 1 and the PWM is for both wheel speeds (pin 16 and 17) are set to 80% duty cycle. If the "adj left" command is given, then the left wheel is stopped by setting the left wheel speed to 0% and the right wheel speed to 80% duty cycle. If the "adj right" command is given, then the right wheel is stopped by setting the right wheel speed to 0% and the left wheel speed to 80% duty cycle. For "hard left" or "hard right" the wheels are turned in opposite directions as appropriate. For "reverse" both wheel directions are set to logic 0 and the PWMs are set to 80% duty cycle.

What's wrong with this?

Module/Class Design Documentation

Interface — How can the module/class be used?

- Methods and their arguments, input and output variables (be specific: data type, interpretation)
- Does it interact with the environment? (User interface, hardware signals.)

Behaviour

- How are the outputs related to the inputs (i.e., **functional** specification)?
- Use cases, user's guide
- 'Actors' may include other modules

Relationships

– Implementation class diagrams.
– 'Uses' relation (i.e., what modules does this module depend on)

Internal design

– Algorithms (pseudo-code, flowchart etc.)
– Internal data structures

Better Module Design Documentation

Name: Steering module

Exported types:

`enum Direction { Forward, AdjLeft, AdjRight, HardLeft, HardRight, Reverse, Stop }`

Methods: `void steer(Direction dir)` — sets the direction of travel to `dir`.

Behaviour:

– Initial state is Stop.
– Each call to `steer` changes the output state to that given by its argument, as follows:

`dir`	Left wheel	Right wheel
Stop	stopped	stopped
Forward	forward, full	forward, full
AdjLeft	stopped	forward, full
AdjRight	forward, full	stopped
HardLeft	reverse, full	forward, full
HardRight	forward, full	reverse, full
Reverse	reverse, full	reverse, full

Steering Module Internal Design

Hardware signals:

Signal	Description	Location
L_Dir	Left wheel direction, 1 = forward, 0 = reverse	PIC:15, HB:2, $\overline{\mathrm{HB}}$:7
R_Dir	Right wheel direction, 1 = forward, 0 = reverse	PIC:18 HB:15, $\overline{\mathrm{HB}}$:10
L_Speed	Left wheel speed, PWM signal	PIC:16 HB:1
R_Speed	Right wheel speed PWM signal	PIC:17 HB:9

Outputs:

State	L_Dir	R_Dir	L_Speed (% duty)	R_Speed (% duty)
Stop	1	1	0	0
Forward	1	1	80	80
AdjLeft	1	1	0	80
AdjRight	1	1	80	0
HardLeft	0	1	80	80
HardRight	1	0	80	80
Reverse	0	0	80	80

Verification & Validation

Verification — Does it do what you said it should do?

Validation — Does it do what the client wants?

- Testing
- Reviews/Inspections

Testing

- Plan your tests in advance (when component interface is designed).
  - Test environment
  - Input
  - Expected results
- The goal is to find errors — try hard to break the system.
- Aim for high coverage.
- Don't forget to test the error cases.

Reviews/Inspections

- Check that a product (e.g., code, module documentation) satisfies it's requirements (without executing it).
- Should be carried out by someone other than the author — ego-less programming.
- Client reviews are very useful for validation.
- Only method to ensure that documentation is correct (unless formal techniques are used).
- For code, inspection has been shown to be <u>much more</u> efficient than testing at finding bugs.