Data Structures and Algorithms for Engineers

Module 1: Introduction

Lecture 1: Levels of abstraction. The software development life cycle. Formalisms for representing algorithms.

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Muḥammad ibn Mūsā al-Khwārizmī
محمد بن موسى الخوارزمي

Born approximately 780, died between 835 and 850
Persian mathematician and astronomer
from the Khorasan province of present-day Uzbekistan

The word algorithm is derived from his name

Algorithms + Data Structures = Programs

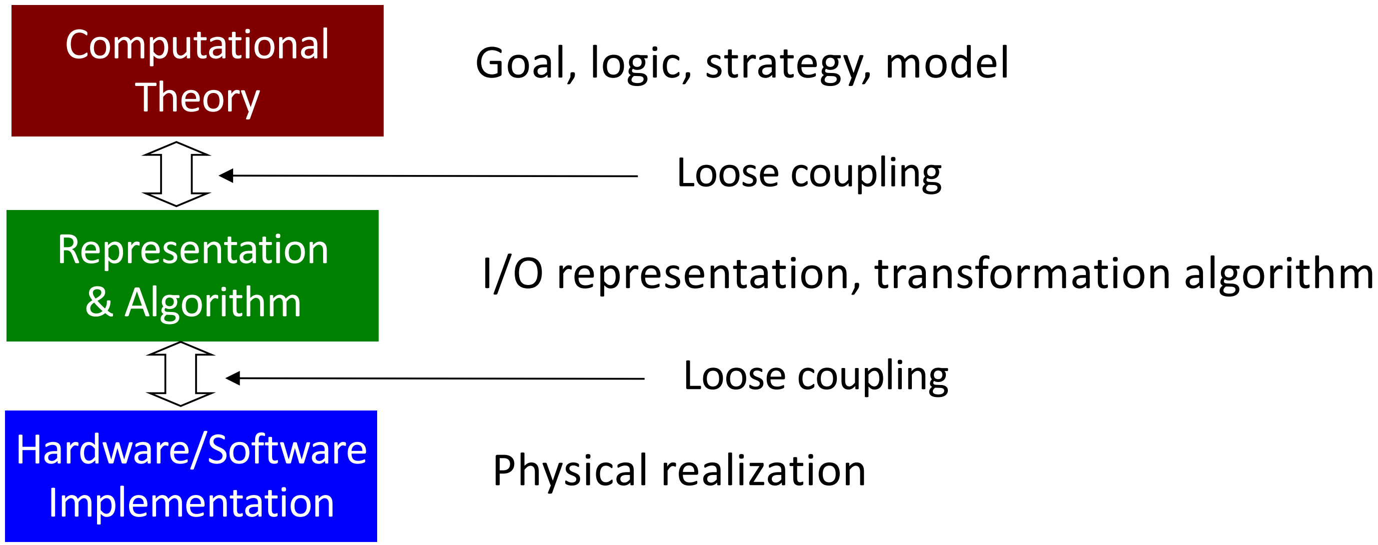
Niklaus Wirth, 1976

Inventor of Pascal and Modula programming languages
Winner of Turing Award 1984


1969

Information Processing:
Representation & Transformation

Marr’s Hierarchy of Abstraction / Levels of Understanding Framework

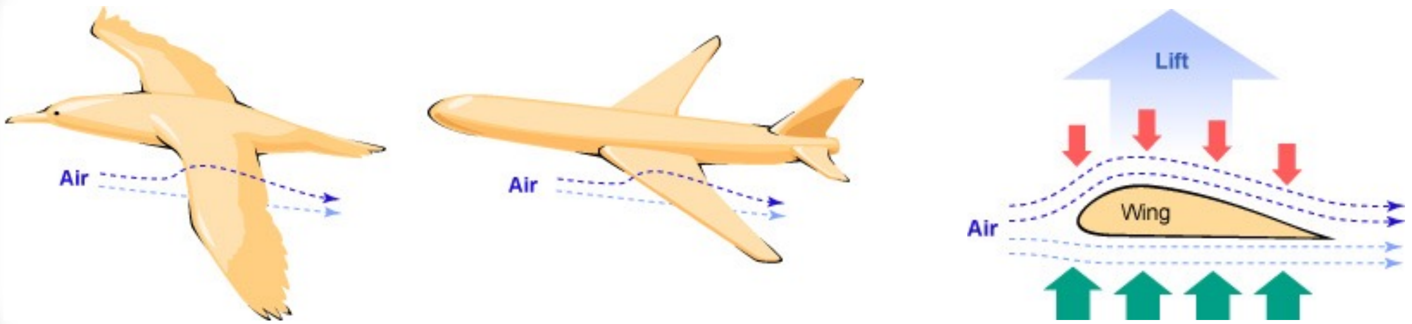


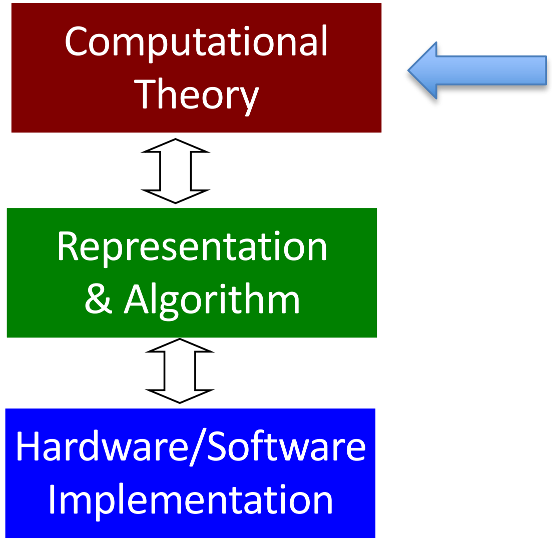
D. Marr and T. Poggio. "From understanding computation to understanding neural circuitry", in E. Poppel, R. Held, and J. E. Dowling, editors, Neuronal Mechanisms in Visual Perception, volume 15 of Neurosciences Research Program Bulletin, pages 470–488. 1977.
D. Marr. Vision. Freeman, San Francisco, 1982.
T. Poggio. The levels of understanding framework, revised. Perception, 41:1017–1023, 2012.

Marr’s Hierarchy of Abstraction / Levels of Understanding Framework

“Trying to understand perception by studying only neurons is like trying to understand bird flight by studying only feathers: it just cannot be done.
In order to understand bird flight, we have to understand aerodynamics;
only then do the structure of feathers and the different shapes of birds’ wings make sense”

Marr, D. *Vision*, Freeman, 1982.



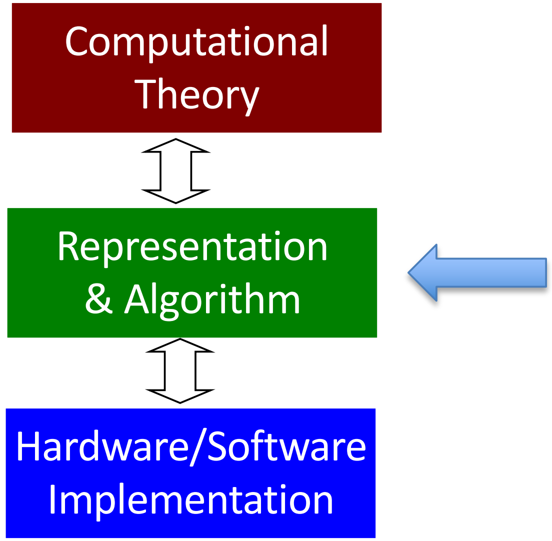


Sorting a List

Given a sequence of *n* keys $a_1, \dots, a_n$

Find the permutation (reordering) such that $a_i \leq a_j$

$1 \leq i, j \leq n$



Sorting a List

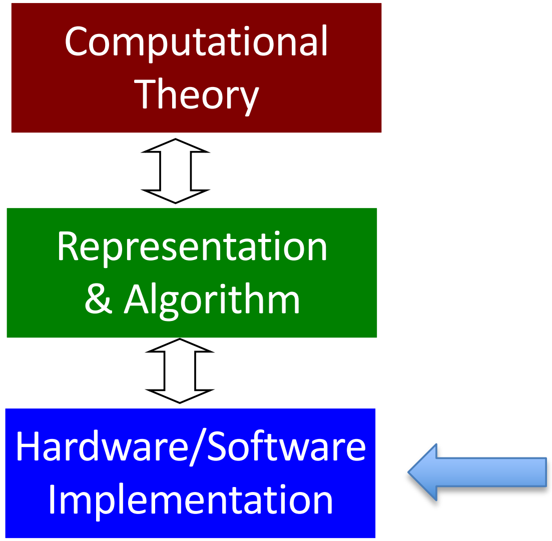
Bubble Sort

Insertion Sort

Quick Sort

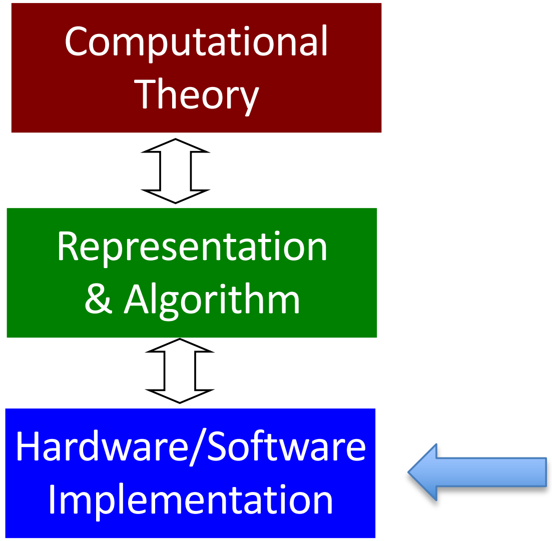
Merge Sort, …

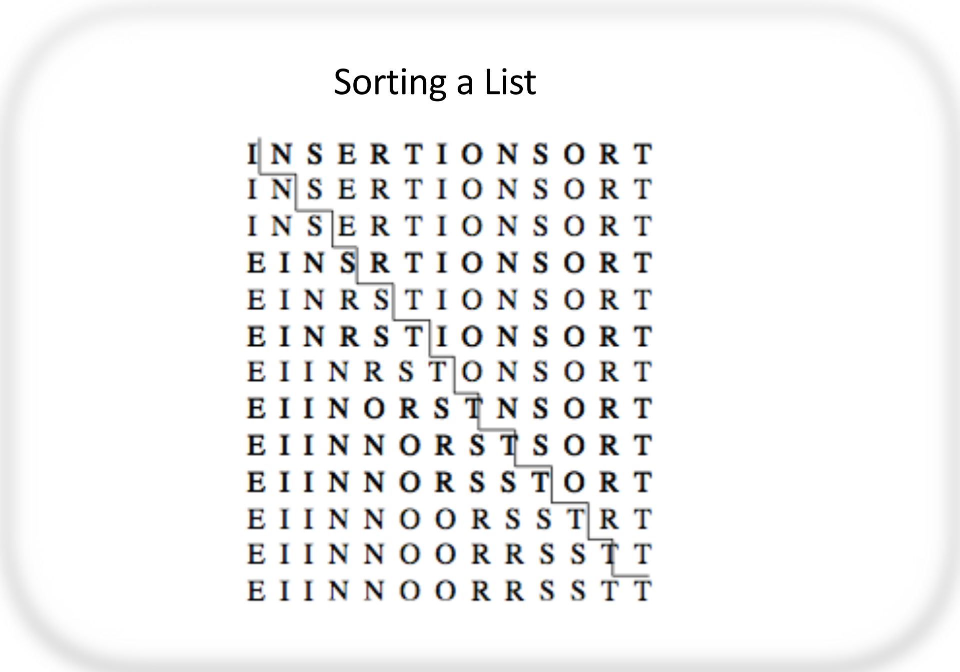
Key point: different computational complexity

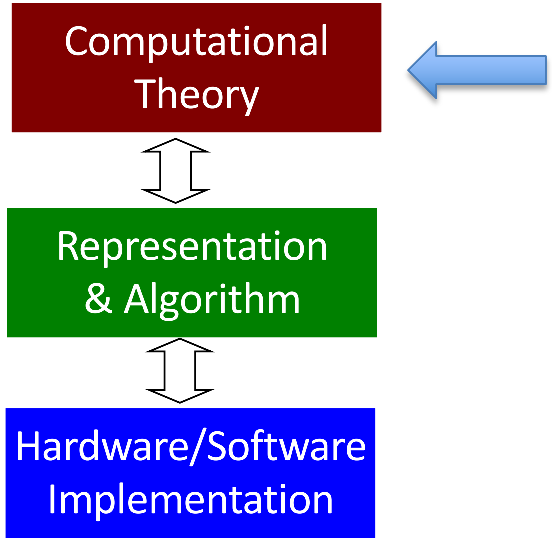


Sorting a List

```
insertion_sort(item s[], int n)
{
        int i,j;                /* counters */

        for (i=1; i<n; i++) {
                j=i;
                while ((j>0) && (s[j] < s[j-1])) {
                        swap(&s[j],&s[j-1]);
                        j = j-1;
                }
        }
}
```


Computational Theory
Representation & Algorithm
Hardware/Software Implementation

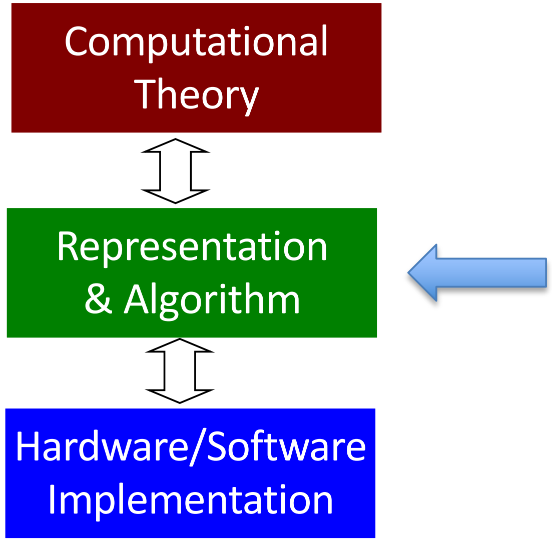

Sorting a List
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T



Fourier Transform

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathsf{F}(\omega_x, \omega_y) \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i(\omega_x x + \omega_y y)}\mathrm{d}x\mathrm{d}y\end{aligned}$$

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathsf{F}(\omega_x, \omega_y) \\ &= \mathsf{F}(\omega_x \Delta_{\omega_x}, \omega_y \Delta_{\omega_y}) \\ &= \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-i(\frac{\omega_x x}{M} + \frac{\omega_y y}{N})}\end{aligned}$$



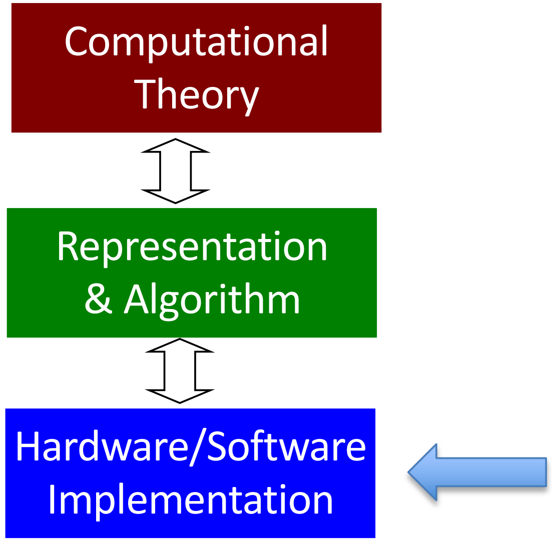
Fourier Transform

DFT: Discrete Fourier Transform

FFT: Fast Fourier Transform

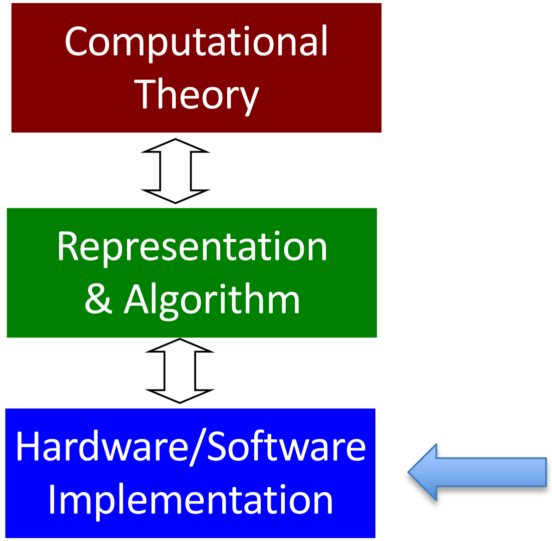
FFTW: Fastest Fourier Transform in the West

Key point: different computational complexity

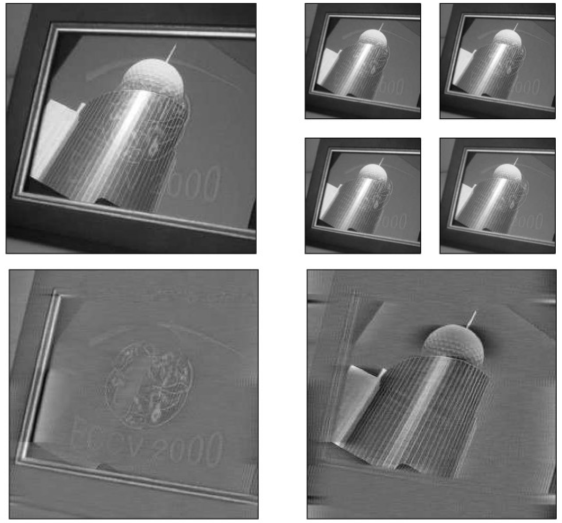


Fourier Transform

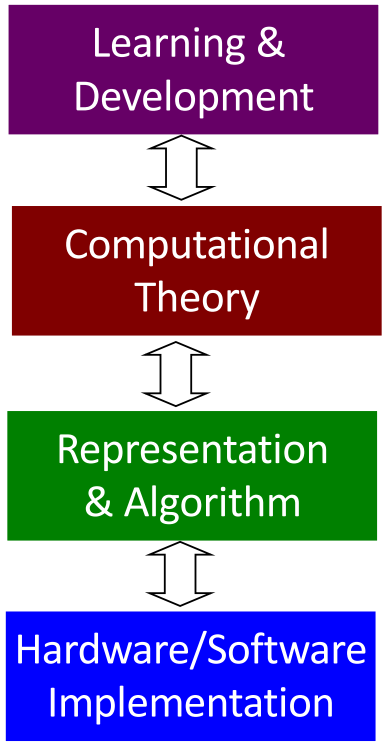
```
main()
{
        unsigned long i;
        int isign;
        float *data1,*data2,*fft1,*fft2;

        data1=vector(1,N);
        data2=vector(1,N);
        fft1=vector(1,N2);
        fft2=vector(1,N2);
        for (i=1;i<=N;i++) {
                data1[i]=floor(0.5+cos(i*2.0*PI/PER));
                data2[i]=floor(0.5+sin(i*2.0*PI/PER));
        }
        twofft(data1,data2,fft1,fft2,N);
        printf("Fourier transform of first function:\n");
        prntft(fft1,N);
        printf("Fourier transform of second function:\n");
        prntft(fft2,N);
        /* Invert transform */
        isign = -1;
        four1(fft1,N,isign);
        printf("inverted transform  =  first function:\n");
        prntft(fft1,N);
        four1(fft2,N,isign);
        printf("inverted transform  =  second function:\n");
        prntft(fft2,N);
        free_vector(fft2,1,N2);
        free_vector(fft1,1,N2);
        free_vector(data2,1,N);
        free_vector(data1,1,N);
        return 0;
}
```



Fourier Transform

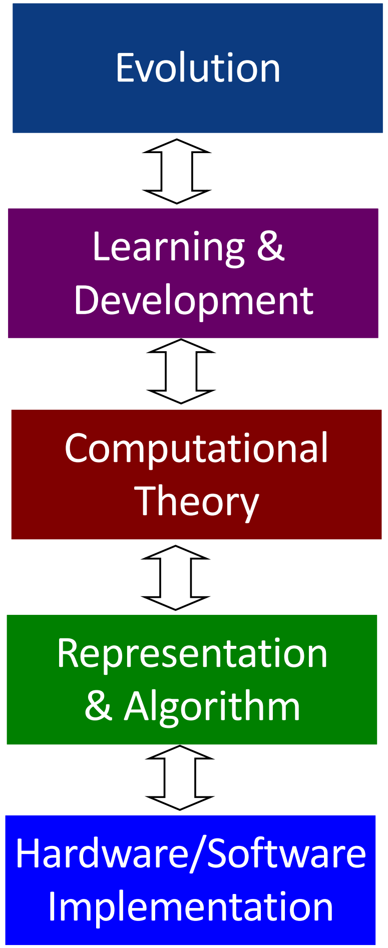


Marr's Levels of Understanding Framework updated 2012 by T. Poggio



Calibrating & improving the model

Marr’s Levels of Understanding Framework updated 2012 by T. Poggio

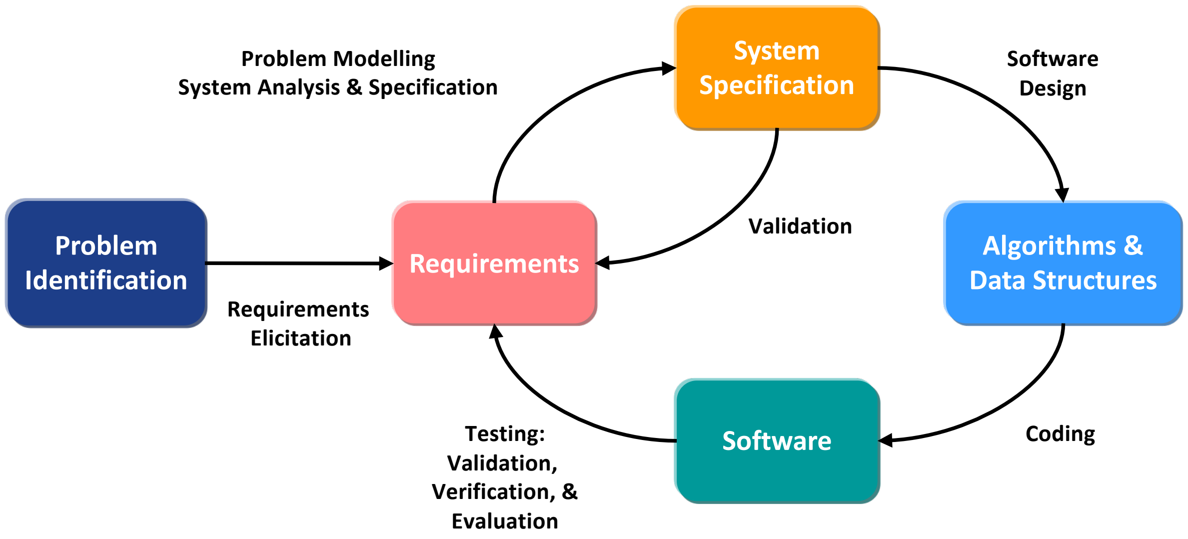


Generating new models

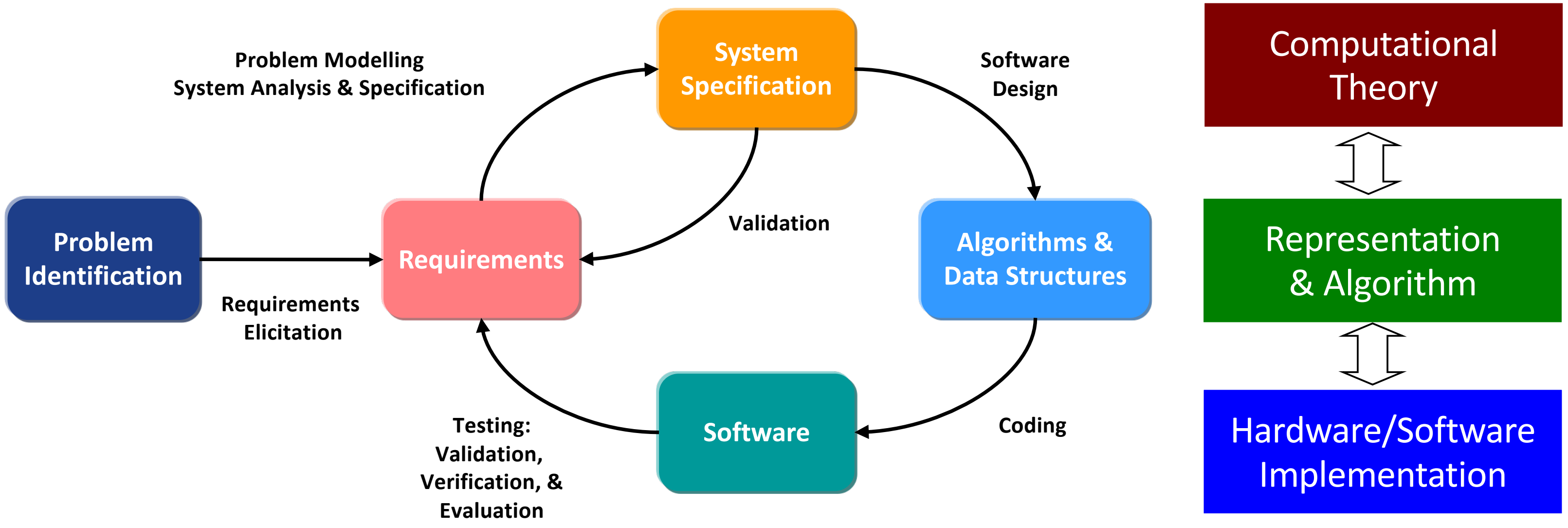
Calibrating & improving the model

Life Cycle Models

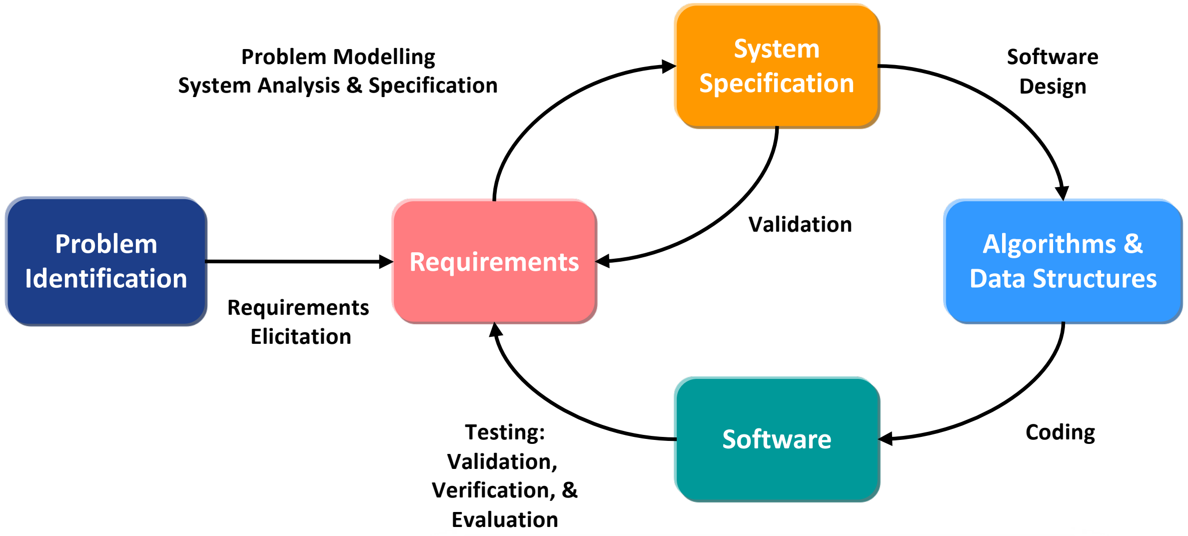
The Software Development Life Cycle

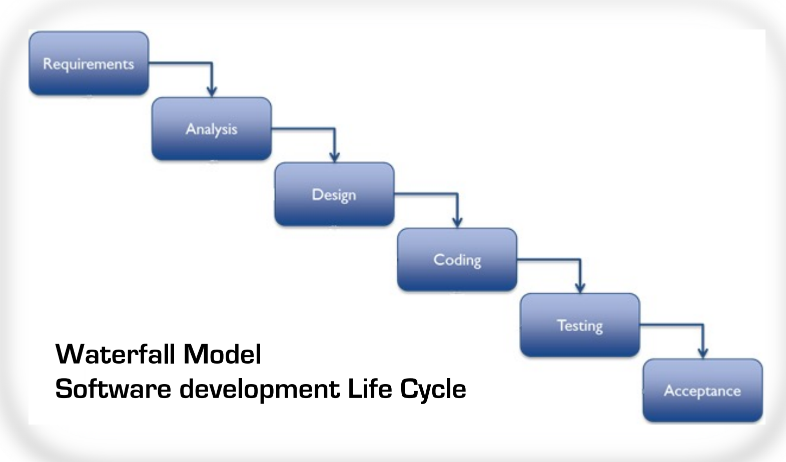


The Software Development Life Cycle

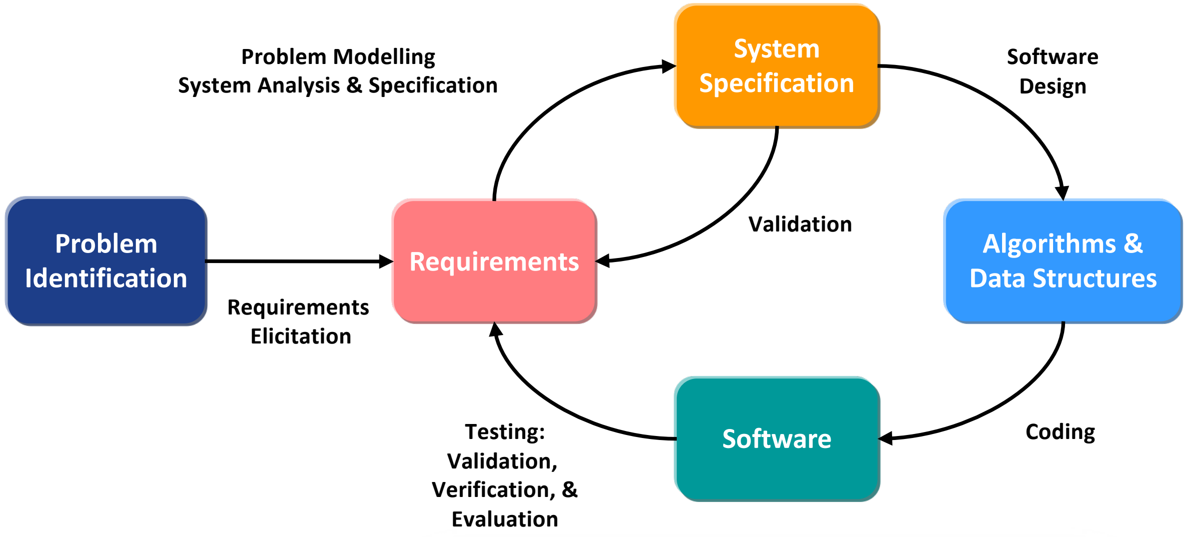


The Software Development Life Cycle





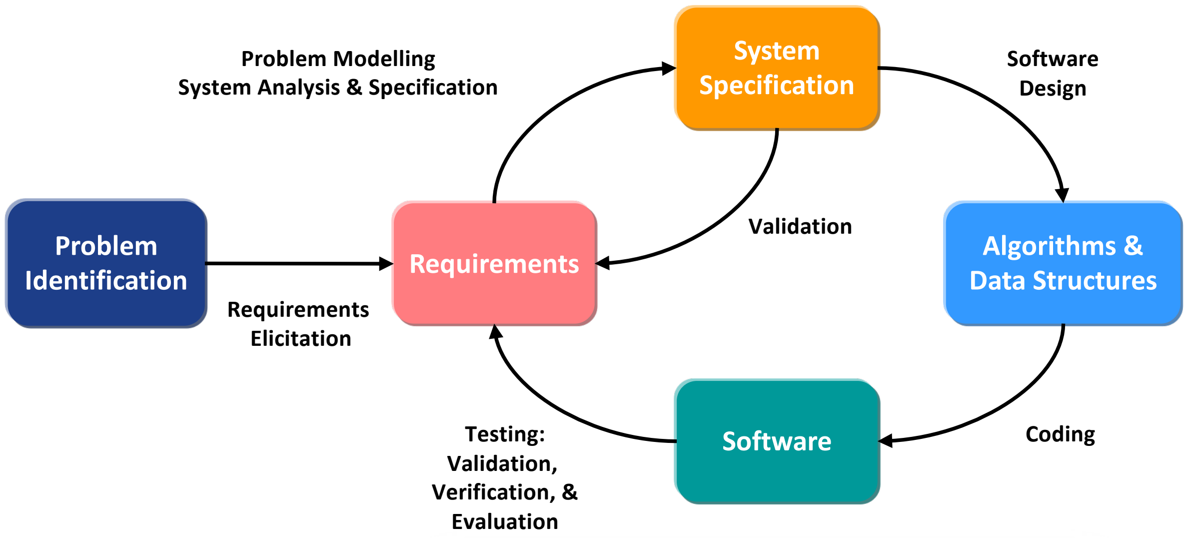
The Software Development Life Cycle



Life Cycle Models (Software Process Models):

Waterfall (& variants, e.g. V)
Evolutionary
Re-use
Hybrid
Spiral
...

The Software Development Life Cycle



Software Development Methodologies:

Structured

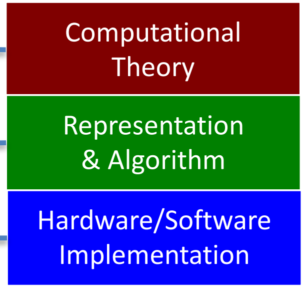
- Yourdon Structured Analysis (YSA)
- Jackson Structured Analysis (JSA)
- Structured Analysis and Design Technique (SADT)

Object-oriented analysis, design, programming

Component-based software engineering (CBSE)

Software Development Life Cycle

1. Problem identification
2. Requirements elicitation
3. Problem modelling
4. System analysis & specification
5. System design
6. Module implementation and system integration
7. System test and evaluation
8. Documentation



Software Development Life Cycle

1. Problem identification

   – Normally requires experience

   – Theoretical issues: appropriate models (problem domain)

   – Technical issues: tools, OS, API, libraries (solution domain)

Software Development Life Cycle

2. Requirements elicitation

   - Talk to the client (by talk, I mean counsel and coach)
   - Document agreed requirements

     What it does, what it doesn't do, how the user is to use it or how it communicates with the user, what messages it displays, how it behaves when the user asks it to do something it expects, and especially how it behaves when the user asks it to do something it doesn't expect

   - Validate requirements with client
   - Repeat until mutual understanding converges
   - But beware …

Software Development Life Cycle

2. Requirements elicitation

Customer to a software engineer:

"I know you believe you understood
what you think I said,
but I am not sure you realize
that what you heard is not what I meant".

R. Pressman

Software Development Life Cycle

3. Problem modelling

   - Identify theory needed to model and solve the problem
     - Ideally, identify several, compare them, and choose the best (i.e., most appropriate)
     - Use criteria derived from your functional and non-functional requirements
   - Create a rigorous – ideally mathematical – description
     Graph theory, Fourier theory, linear system theory, information theory, ...
   - If you don't have a model, you aren't doing engineering
     - Connecting components (or lines of code) together is not engineering
     - Without a model, you can't analyze the system and make firm statement about
       - Robustness
       - Operating parameters
       - Limitations

Software Development Life Cycle

4. System analysis & specification

   - Identify
     - The system functionality
     - The operational parameters (conditions under which your system will operate, including required software and hardware systems)
     - Limitations & restrictions
     - User interface or system interface

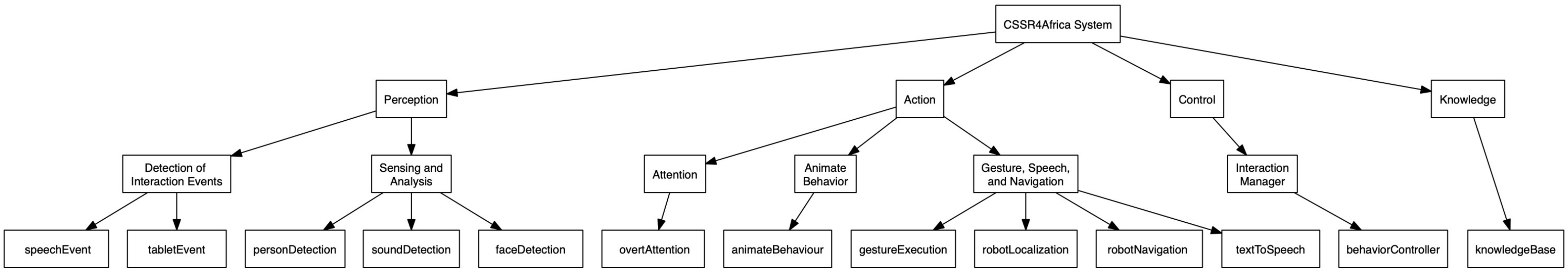
   - Including
     - Functional model
     - Data model
     - Process-flow model
     - Behavioural model

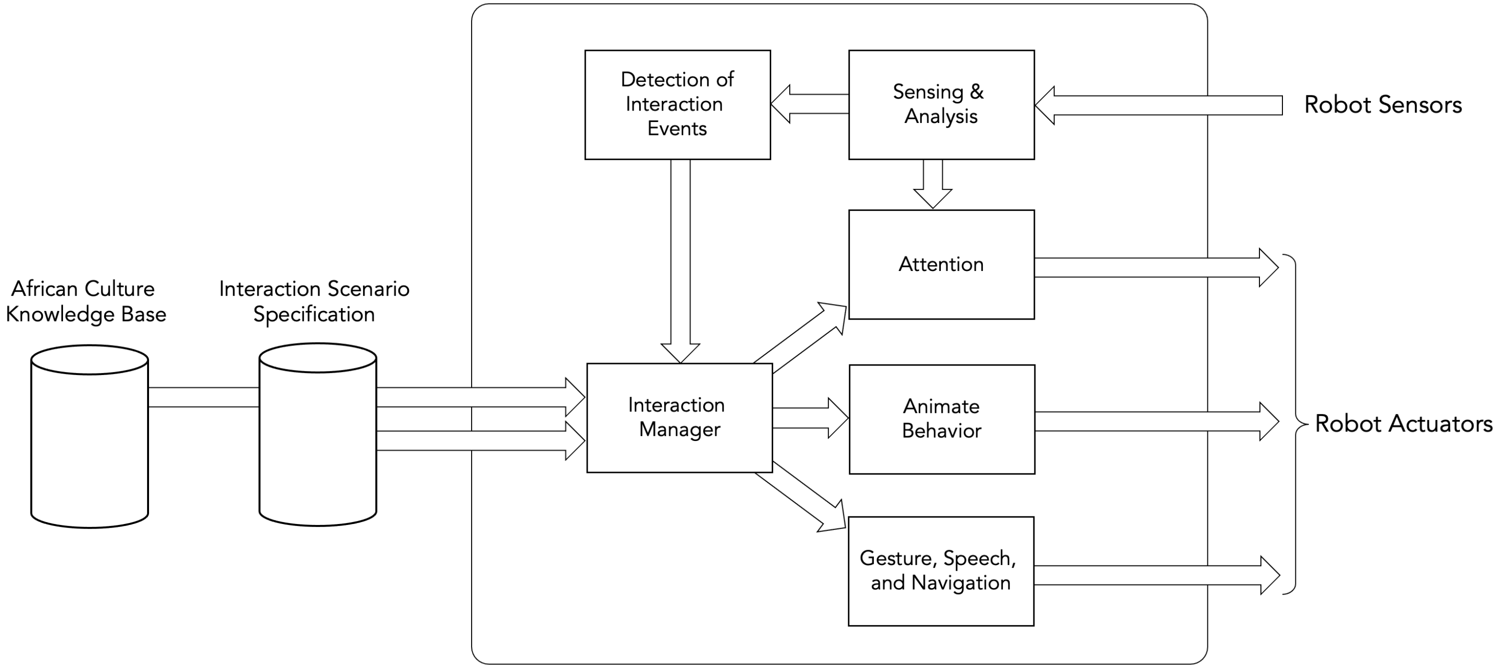
Software Development Life Cycle

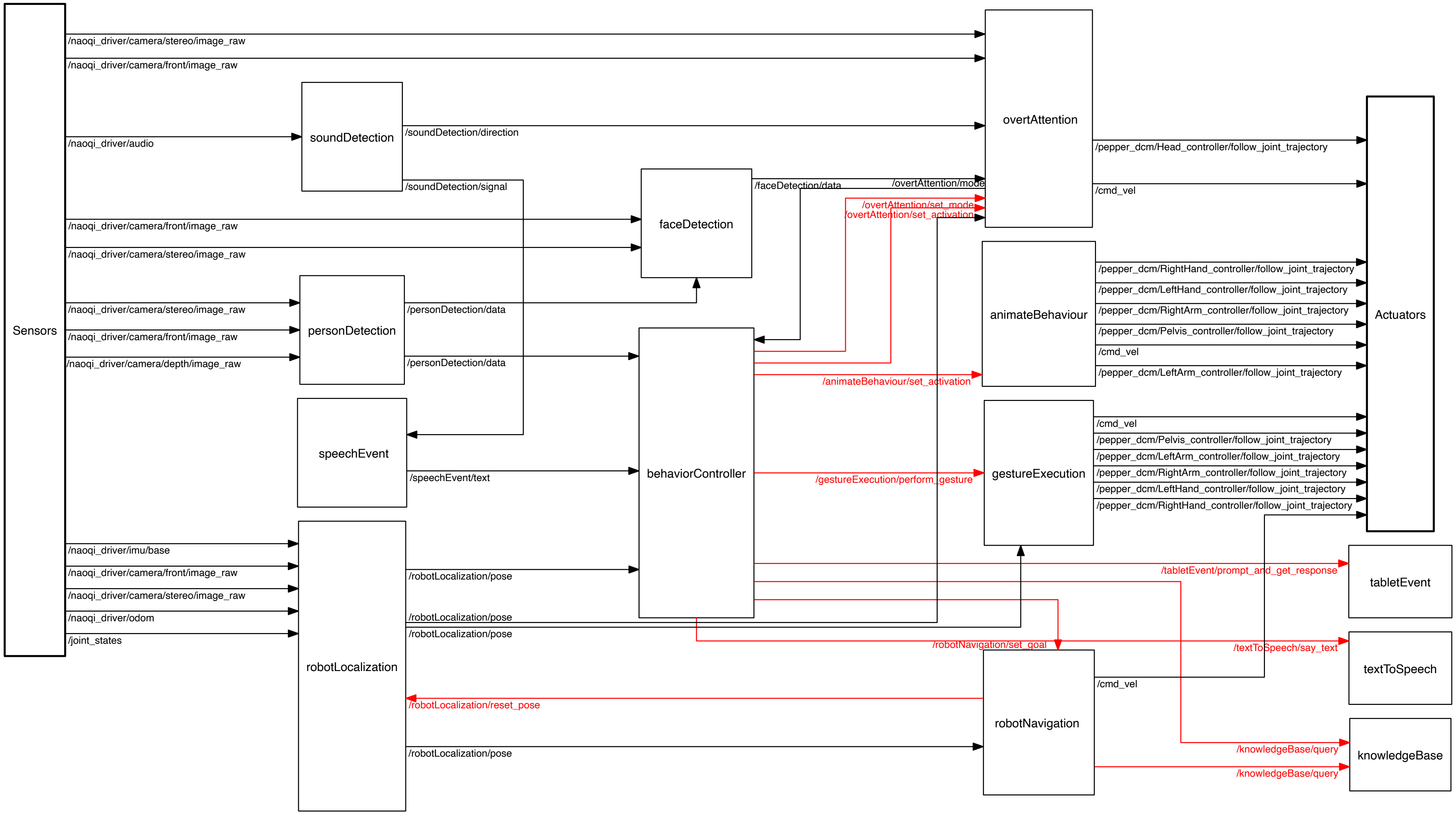
4. System analysis & specification

   Functional model

   - Hierarchical functional decomposition tree
   - Modular decomposition (typically)
   - Each leaf node in the tree:
     - Short description of functionality, i.e. the input/output transformation
     - Information (data) input
     - Information (data) output
   - System architecture diagram
     - Network of components at first or second level of decomposition


CSSR4Africa System
Perception
Action
Control
Knowledge
Detection of Interaction Events
Sensing and Analysis
Attention
Animate Behavior
Gesture, Speech, and Navigation
Interaction Manager
speechEvent
tabletEvent
personDetection
soundDetection
faceDetection
overtAttention
animateBehaviour
gestureExecution
robotLocalization
robotNavigation
textToSpeech
behaviorController
knowledgeBase


Detection of
Interaction
Events
Sensing &
Analysis
Robot Sensors
Attention
African Culture
Knowledge Base
Interaction Scenario
Specification
Interaction
Manager
Animate
Behavior
Robot Actuators
Gesture, Speech,
and Navigation



Sensors
/naoqi_driver/camera/stereo/image_raw
/naoqi_driver/camera/front/image_raw
/naoqi_driver/audio
soundDetection
/soundDetection/direction
/soundDetection/signal
overtAttention
/pepper_dcm/Head_controller/follow_joint_trajectory
/cmd_vel
/faceDetection/data
/overtAttention/mode
/overtAttention/set_mode
/overtAttention/set_activation
/naoqi_driver/camera/front/image_raw
/naoqi_driver/camera/stereo/image_raw
faceDetection
/naoqi_driver/camera/stereo/image_raw
/naoqi_driver/camera/front/image_raw
/naoqi_driver/camera/depth/image_raw
personDetection
/personDetection/data
/personDetection/data
animateBehaviour
/pepper_dcm/RightHand_controller/follow_joint_trajectory
/pepper_dcm/LeftHand_controller/follow_joint_trajectory
/pepper_dcm/RightArm_controller/follow_joint_trajectory
/pepper_dcm/Pelvis_controller/follow_joint_trajectory
/cmd_vel
/pepper_dcm/LeftArm_controller/follow_joint_trajectory
Actuators
/animateBehaviour/set_activation
speechEvent
/speechEvent/text
behaviorController
/gestureExecution/perform_gesture
gestureExecution
/cmd_vel
/pepper_dcm/Pelvis_controller/follow_joint_trajectory
/pepper_dcm/LeftArm_controller/follow_joint_trajectory
/pepper_dcm/RightArm_controller/follow_joint_trajectory
/pepper_dcm/LeftHand_controller/follow_joint_trajectory
/pepper_dcm/RightHand_controller/follow_joint_trajectory
/naoqi_driver/imu/base
/naoqi_driver/camera/front/image_raw
/naoqi_driver/camera/stereo/image_raw
/naoqi_driver/odom
/joint_states
robotLocalization
/robotLocalization/pose
/robotLocalization/pose
/robotLocalization/pose
/tabletEvent/prompt_and_get_response
tabletEvent
/robotNavigation/set_goal
/textToSpeech/say_text
textToSpeech
/cmd_vel
robotNavigation
/robotLocalization/reset_pose
/robotLocalization/pose
/knowledgeBase/query
/knowledgeBase/query
knowledgeBase

Software Development Life Cycle

4. System analysis & specification

Modular decomposition … Dave Parnas

“In this context "module" is considered to be a responsibility assignment rather than a subprogram. The modularizations include the design decisions which must be made before the work on independent modules can begin.”

D.L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12, Dec 1972

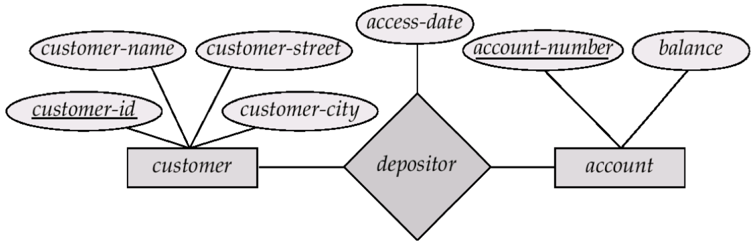
Also responsible for the concepts of data hiding and encapsulation, cf. ADTs in Lecture DSA02-04

Software Development Life Cycle

4. System analysis & specification

Data model

- Data entities (not data structures) to represent
  - Input, temporary, output data
- Data dictionary
  - What the data entities mean
  - How they are composed
  - How they are structured
  - Valid value ranges
  - Dimensions (e.g., velocity m/s)
  - Relationships between data entities
- Entity-relationship model



Software Development Life Cycle

4. System analysis & specification

   Process-flow model

   – What data flows into and out of each functional block
     (into and out of the leaf nodes in the functional decomposition tree)

   – Data-flow diagrams
     - Organized in several levels: DFD level 0, DFD level 1, …
     - Level 0 DFD: system architecture diagram

Software Development Life Cycle

4. System analysis & specification

   Process-flow model

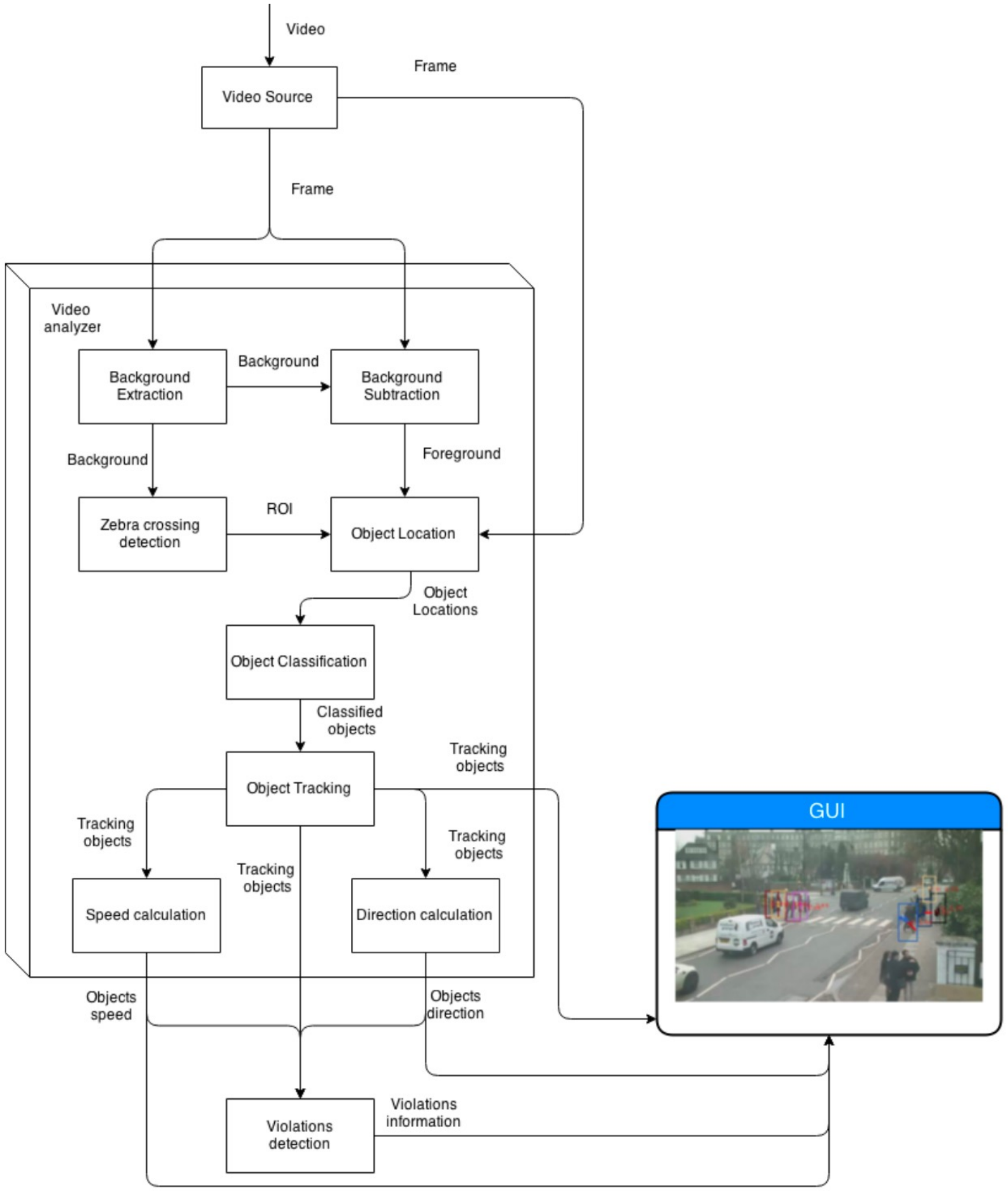
   – DFDs model the transformation of inputs into outputs

   – Processes/Functions represent individual functions that the system carries out and transform inputs to outputs

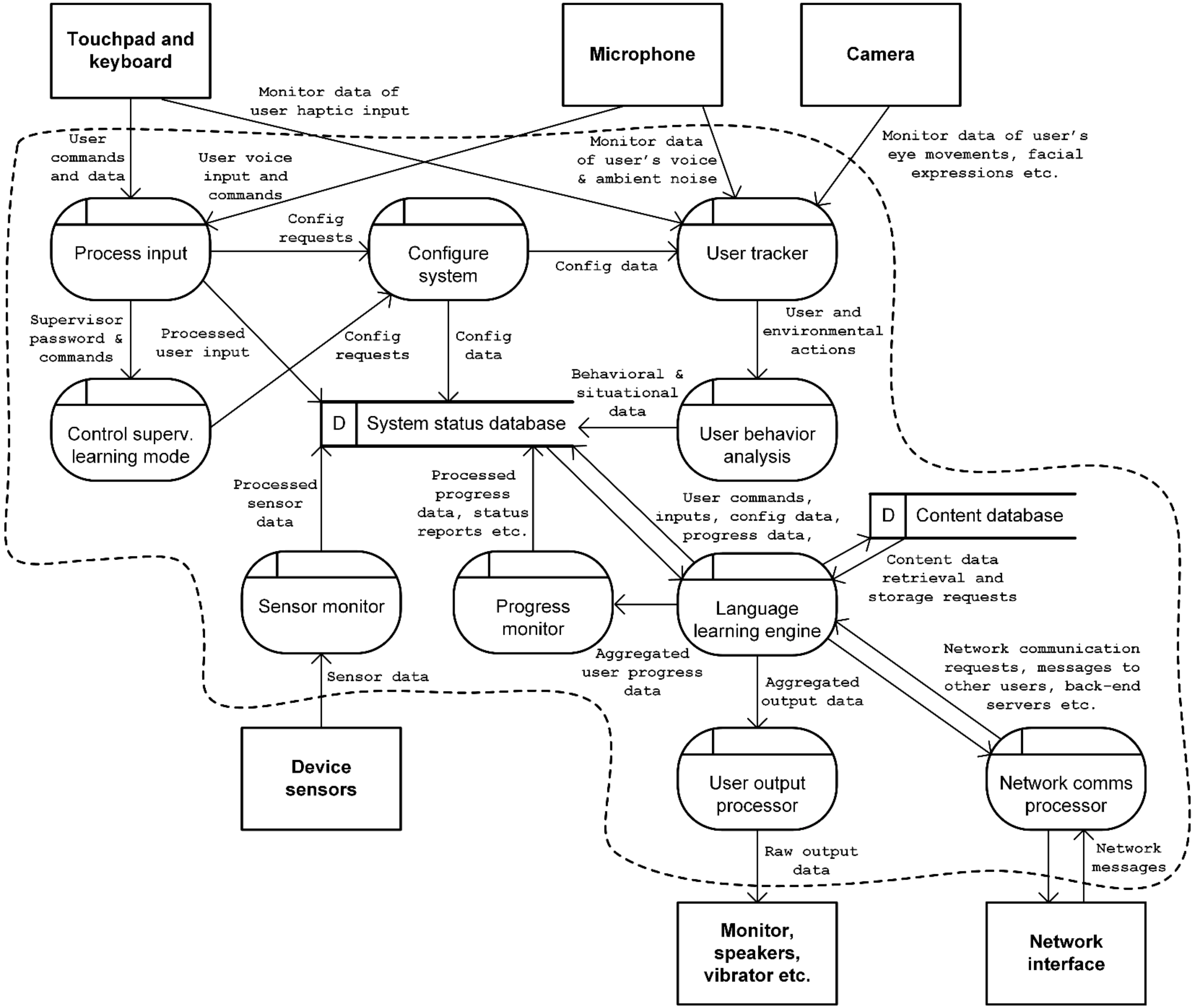
   – Flows represent connections between processes and the flow of information and data between processes

   – Data Stores show collections or aggregations of data

   – I/O Entities show external entities with which the system communicates
     - They are the sources and consumers of data
     - They can be users, groups, organizations, systems,...



Video
Video Source
Frame
Frame
Video analyzer
Background Extraction
Background
Background Subtraction
Background
Foreground
Zebra crossing detection
ROI
Object Location
Object Locations
Object Classification
Classified objects
Object Tracking
Tracking objects
Tracking objects
Tracking objects
Tracking objects
Speed calculation
Direction calculation
GUI
Objects speed
Objects direction
Violations detection
Violations information



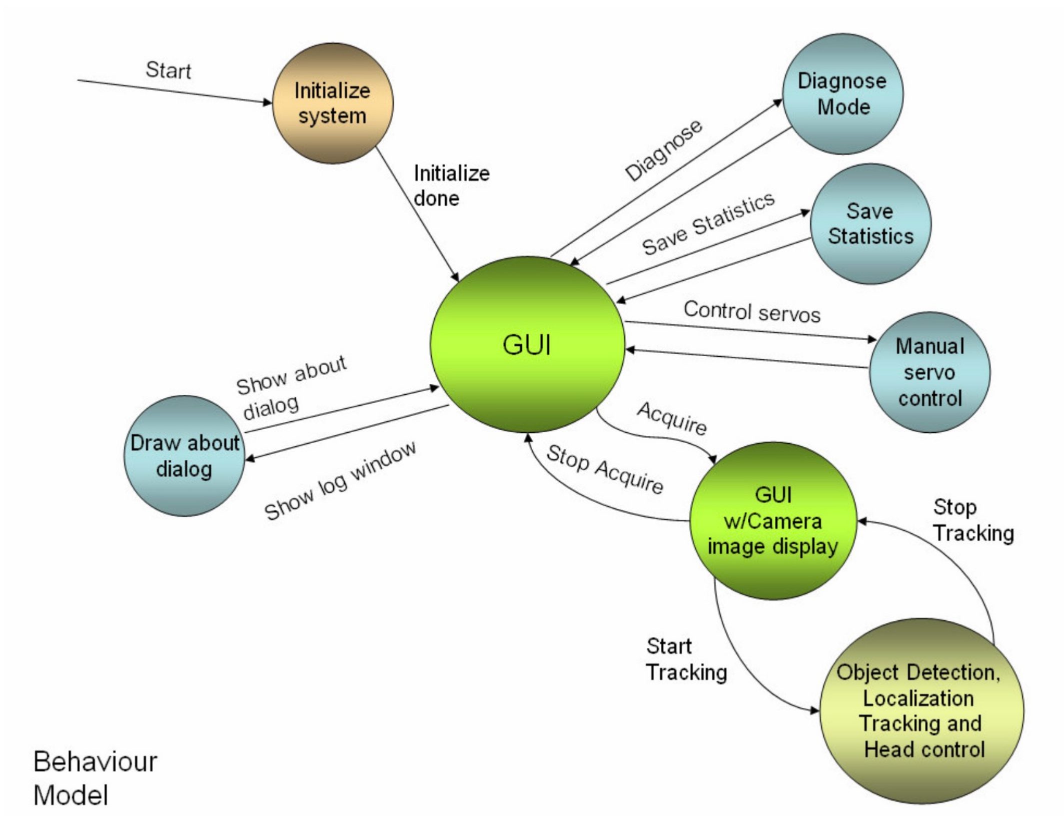
Touchpad and keyboard
Microphone
Camera
Monitor data of user haptic input
User commands and data
User voice input and commands
Monitor data of user's voice & ambient noise
Monitor data of user's eye movements, facial expressions etc.
Process input
Config requests
Configure system
Config data
User tracker
Supervisor password & commands
Processed user input
Config requests
Config data
User and environmental actions
Behavioral & situational data
Control superv. learning mode
D
System status database
User behavior analysis
Processed sensor data
Processed progress data, status reports etc.
User commands, inputs, config data, progress data,
D
Content database
Content data retrieval and storage requests
Sensor monitor
Progress monitor
Language learning engine
Aggregated user progress data
Network communication requests, messages to other users, back-end servers etc.
Sensor data
Aggregated output data
Device sensors
User output processor
Network comms processor
Raw output data
Network messages
Monitor, speakers, vibrator etc.
Network interface

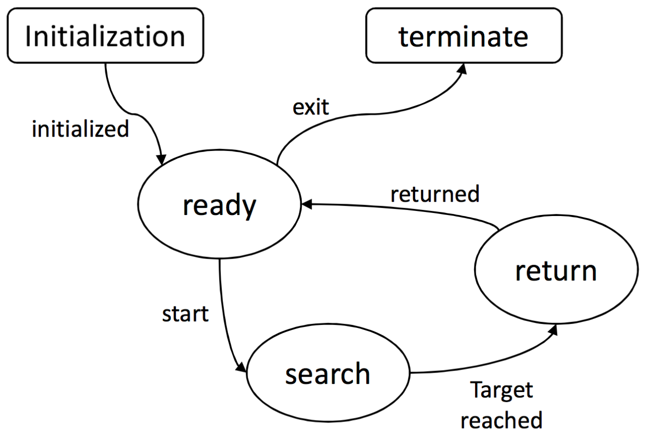
Software Development Life Cycle

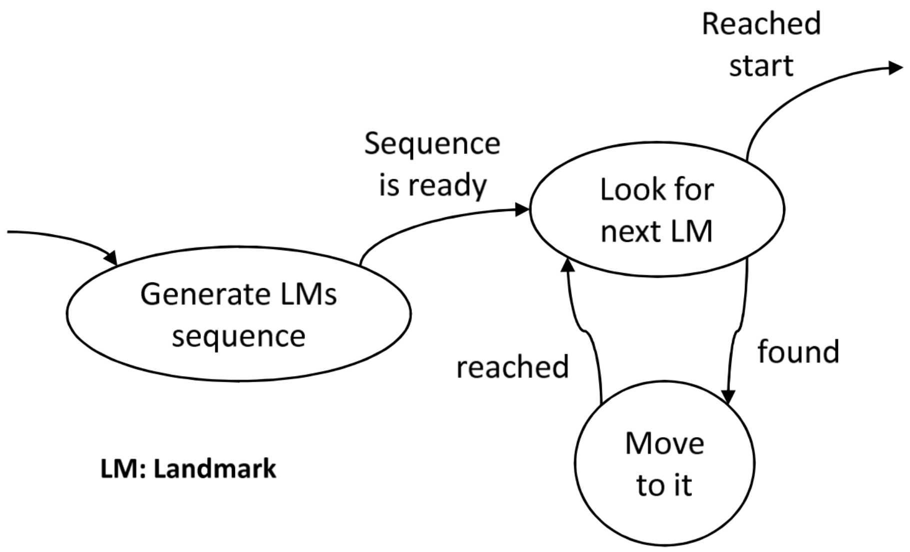
4. System analysis & specification

Behavioural model

- Behaviour over time
- System states
- Triggers that cause transition
  (from state to state)
- Functional block associated with each state
- State transition diagram
  - Finite state machine
  - Finite automaton
- Control-flow diagram
  (version of DFD with events and triggers on each process)


Start
Initialize system
Initialize done
GUI
Diagnose
Diagnose Mode
Save Statistics
Save Statistics
Control servos
Manual servo control
Show about dialog
Draw about dialog
Show log window
Acquire
Stop Acquire
GUI w/Camera image display
Stop Tracking
Start Tracking
Object Detection, Localization Tracking and Head control
Behaviour Model


Initialization
terminate
initialized
exit
ready
returned
return
start
search
Target
reached


Reached
start
Sequence
is ready
Look for
next LM
Generate LMs
sequence
reached
found
Move
to it
LM: Landmark

Software Development Life Cycle

4. System analysis & specification

   Definition of all the user and system interfaces

   – User manual
   – User interface storyboard

Software Development Life Cycle

4. System analysis & specification

   Specification of non-functional characteristics

   – Dependability
   – Security
   – Composability
   – Portability
   – Reusability
   – Interoperability

Often reflect the quality of the system

Software Development Life Cycle

5. Software design

   - For each module (i.e., leaf node in the hierarchical decomposition tree / system architecture diagram / lowest level DFD)

   - Identify several design options & compare them

     - Algorithms ← Effect the functional input-output transformation, i.e., realize computational theory
     - Data-structures ← Representation of the input, temporary, and output data
     - Files
     - Interface protocols

   - Choose the best design

     - You have to define what 'best' means for your particular project

     - Use criteria derived from the functional and non-functional requirements

Software Development Life Cycle

6. Module implementation and system integration

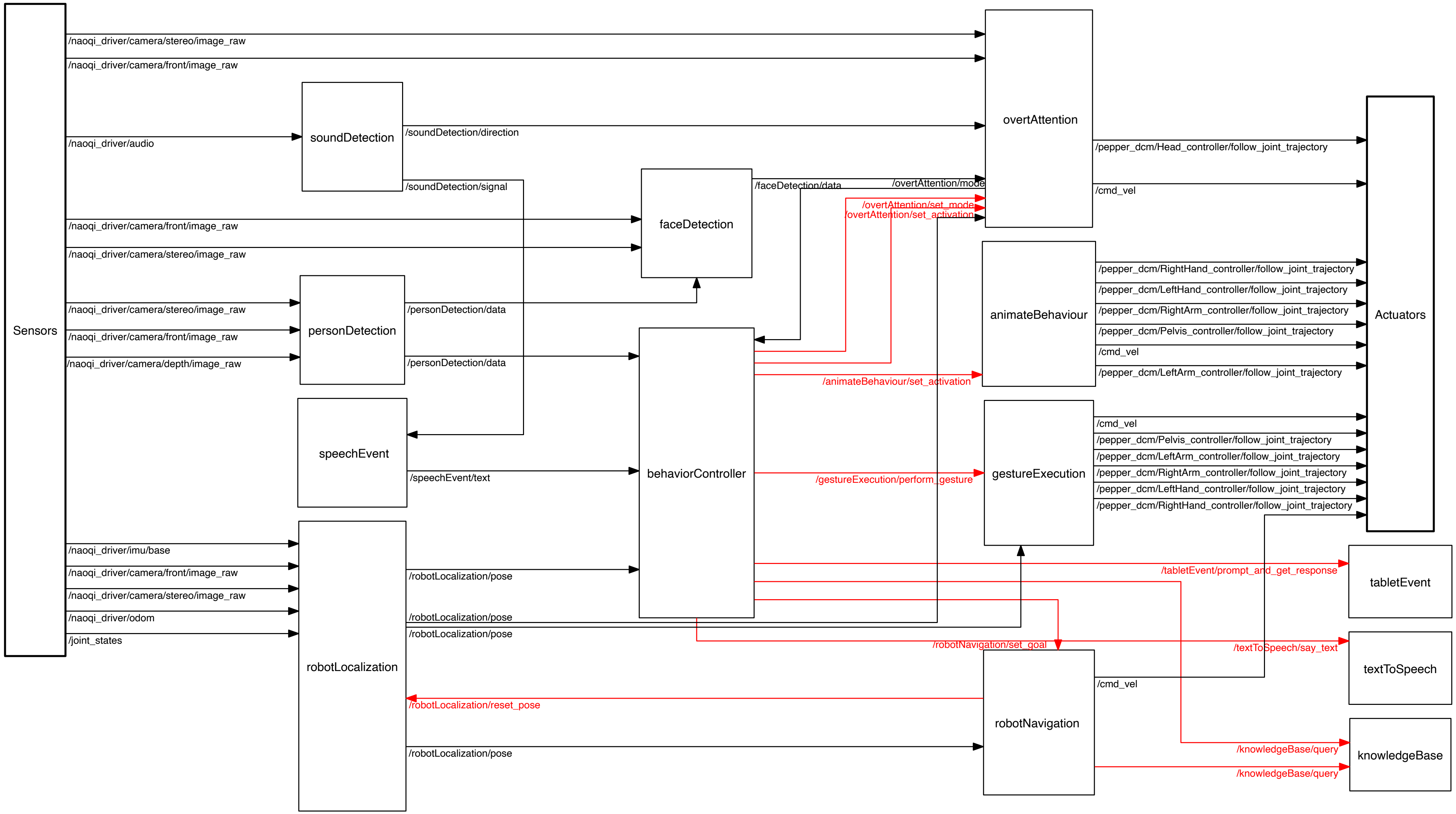
   – Use a modular construction approach

   – Don’t attempt the so-called Big Bang approach

   – Build (and test) each component or modular sub-system individually

     • Driver (dummy calling routine) … test harness
     • Stub (dummy called routine)

   – Link or connect them together, one component at a time.


Sensors
/naoqi_driver/camera/stereo/image_raw
/naoqi_driver/camera/front/image_raw
soundDetection
/naoqi_driver/audio
/soundDetection/direction
/soundDetection/signal
overtAttention
/pepper_dcm/Head_controller/follow_joint_trajectory
/cmd_vel
faceDetection
/faceDetection/data
/overtAttention/mode
/overtAttention/set_mode
/overtAttention/set_activation
personDetection
/naoqi_driver/camera/depth/image_raw
/personDetection/data
animateBehaviour
/pepper_dcm/RightHand_controller/follow_joint_trajectory
/pepper_dcm/LeftHand_controller/follow_joint_trajectory
/pepper_dcm/RightArm_controller/follow_joint_trajectory
/pepper_dcm/Pelvis_controller/follow_joint_trajectory
/pepper_dcm/LeftArm_controller/follow_joint_trajectory
Actuators
/animateBehaviour/set_activation
speechEvent
/speechEvent/text
behaviorController
/gestureExecution/perform_gesture
gestureExecution
/naoqi_driver/imu/base
/naoqi_driver/odom
/joint_states
robotLocalization
/robotLocalization/pose
/tabletEvent/prompt_and_get_response
tabletEvent
/robotNavigation/set_goal
/textToSpeech/say_text
textToSpeech
robotNavigation
/robotLocalization/reset_pose
/knowledgeBase/query
knowledgeBase

Software Development Life Cycle

6. Module implementation and system integration

   You Must Validate Data

   – Validate input

   – Validate parameters

   – 'Constraints on data and computation usually take the form of wrappers – access routines (or methods) that prevent bad data from being stored or used and ensure that all programs modify data through a single, common interface'

     J. A. Whittaker and S. Atkin, "Software Engineering Is Not Enough", IEEE Software, July/August 2002, pp. 108-115.

Software Development Life Cycle

7. Unit, integration, & acceptance test and evaluation

   - NOT about showing the system works
   - Showing it meets specifications
   - Showing it meets requirements
   - Showing the system doesn't fail (stress testing)
   - Three goals of testing

     1. Verification
     2. Validation
     3. Evaluation

Software Development Life Cycle

7. System test and evaluation

   1. Verification

      - Has the system been built correctly?
      - Is it computing the right answer (producing correct data)?
      - Extensive test data sets
      - Exercise each module or computation
        - Independently
        - As a whole system
      - Live data (not just data in test files)

Software Development Life Cycle

7. System test and evaluation

    2. Validation

        - Does it meet the client’s requirements?
        - Can the user adjust all the main parameters on which operation depends? (List them!)

Software Development Life Cycle

7. System test and evaluation

    3. Evaluation

        - How good is the system?
        - Hallmark of good engineering: assess performance and benchmark against other systems
        - Identify quantitative metrics
        - Identify qualitative metrics
        - Vary parameters and collect statistics
        - Evaluate against ground-truth data (data for which you know the correct result)
        - Evaluate against other systems (benchmarking)

Software Development Life Cycle

7. System test and evaluation

   – Tests need to be automated (run several times as the system is tuned)

   – Regression testing

   – Types of test
     - Unit Tests … individual modules / components
     - Integration Tests … sub-systems and system
     - Acceptance Tests … system

Software Development Life Cycle

8. Documentation

- Internal documentation
  - Documentation comments
    - Intended to be extracted automatically by, e.g., Doxygen tool
    - Describe the functionality from an implementation-free perspective
    - Purpose is to explain how to use the component through its application programming interface (API), rather than understand its implementation
  - Implementation comments
    - Overviews of code
    - Provide additional information that is not readily available in the code itself
    - Comments should contain only information that is relevant to reading and understanding the program
  - Use standards

Software Development Life Cycle

8. Documentation

"There is rarely such a thing as too much documentation …

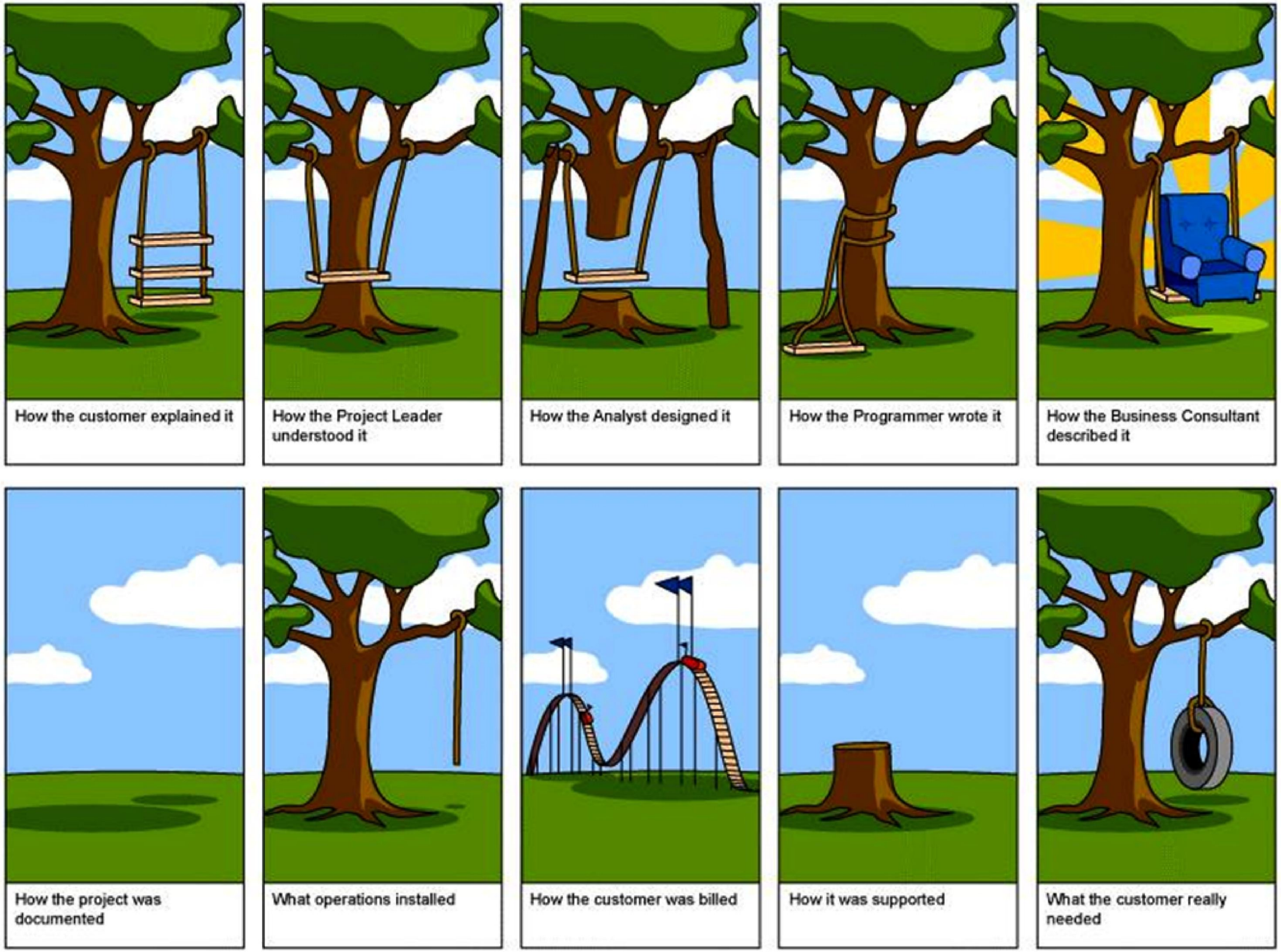
Documentation – often exceeding the source code in size – is a requirement, not an option."

J. A. Whittaker and S. Atkin, "Software Engineering Is Not Enough", IEEE Software, July/August 2002, pp. 108-115.

Software Development Life Cycle

8. Documentation

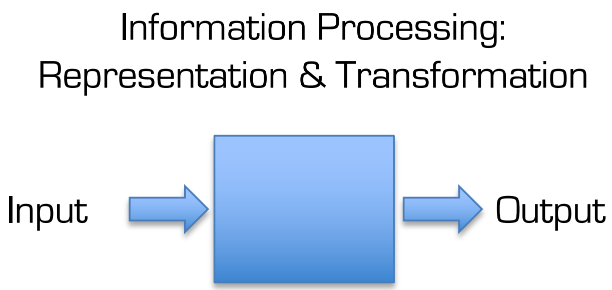
   - External documentation
     - User manual
     - Reference manual
     - Design documents


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Formalisms for Representing Algorithms

Informal definition

An algorithm is a systematic procedure for transforming information from one (input) state to another (output) state



Definition of an Algorithm

Typically, there is a strong link between an algorithm and the information representation, i.e., the data structure

Formalisms for Representing Algorithms

Required characteristics

- Simple, clear, and intuitive (as far as possible)
- As rigorous as practical – but keeping the math as simple as possible
- Language neutral
- Factor out the hardware and operating systems
- Focus on algorithmic essence
- Properly scoped (not too big, not too trivial or obvious)

Practical Representations

- Some candidate representations
  - Pseudo Code
  - Flow Charts
  - State Diagrams
  - Formalisms
  - Modeling Methodologies (e.g., UML)

- Many engineers use these, but some use them
  - At the wrong time
  - To model the wrong kinds of things (poor scoping)
  - Incorrectly
  - Mix “what is needed” with “how we will build it”

Pseudo Code

- Pseudo code is an informal abstraction of an algorithm that:
  - uses the structural conventions of a programming language
  - is simplified for human reading rather than machine compilation
  - omits details that are not essential for algorithmic analysis
  - shows the temporal relation of instruction execution (sequencing)
- Despite many attempts, no standard for pseudo code syntax currently exists

Pseudo Code

Declaration

```
type variable;
integer A; string name;
```

Assignment

```
variable = value;
a = 45; x = y
```

Basic mathematical operators

```
result = variable_value operator variable_value
y = a+b; z = 5.0/e; j = k*l; r = 2*(22/7)*(r^2)
```

Basic functions, subroutines, methods

```
read(), write(), print(),…
```

Assumed functions should be clearly defined prior to use; more on functions, subroutines, and methods later

Pseudo Code

Control Structures

– Direct sequence

  `do X, then do Y`

– Conditional branching

  if Q then do X, else do Y

– Bounded iteration

  do Z exactly X times

– Conditional or unbounded iteration

  do Z until Q becomes true

  while Q is true do Z

Pseudo Code

Example: algorithm to find the greatest common denominator (GCD)

- How the **`read()`** function work is not important for our analysis
- We focus on the essence of the algorithm, not on checking input, formatting output, error handling, and so forth
- Now that the algorithm has been distilled to its essence we can analyze: how do we know we solved the problem? how quickly does it compute the answer?

```
a = read()
b = read()
if a = 0
    return b
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

Pseudo Code

- Pseudo code is attractive because
  - It looks like the computer-interpretable code
  - It is complete in terms of describing computer algorithms
- In practice, pseudo code is sometimes extended and violates notions of minimalism
  - Pseudo code should only support what is necessary to describe the algorithm – and no more!
  - Sometimes, pseudo code is used to describe entire applications, and becomes too cumbersome to support analysis of algorithms

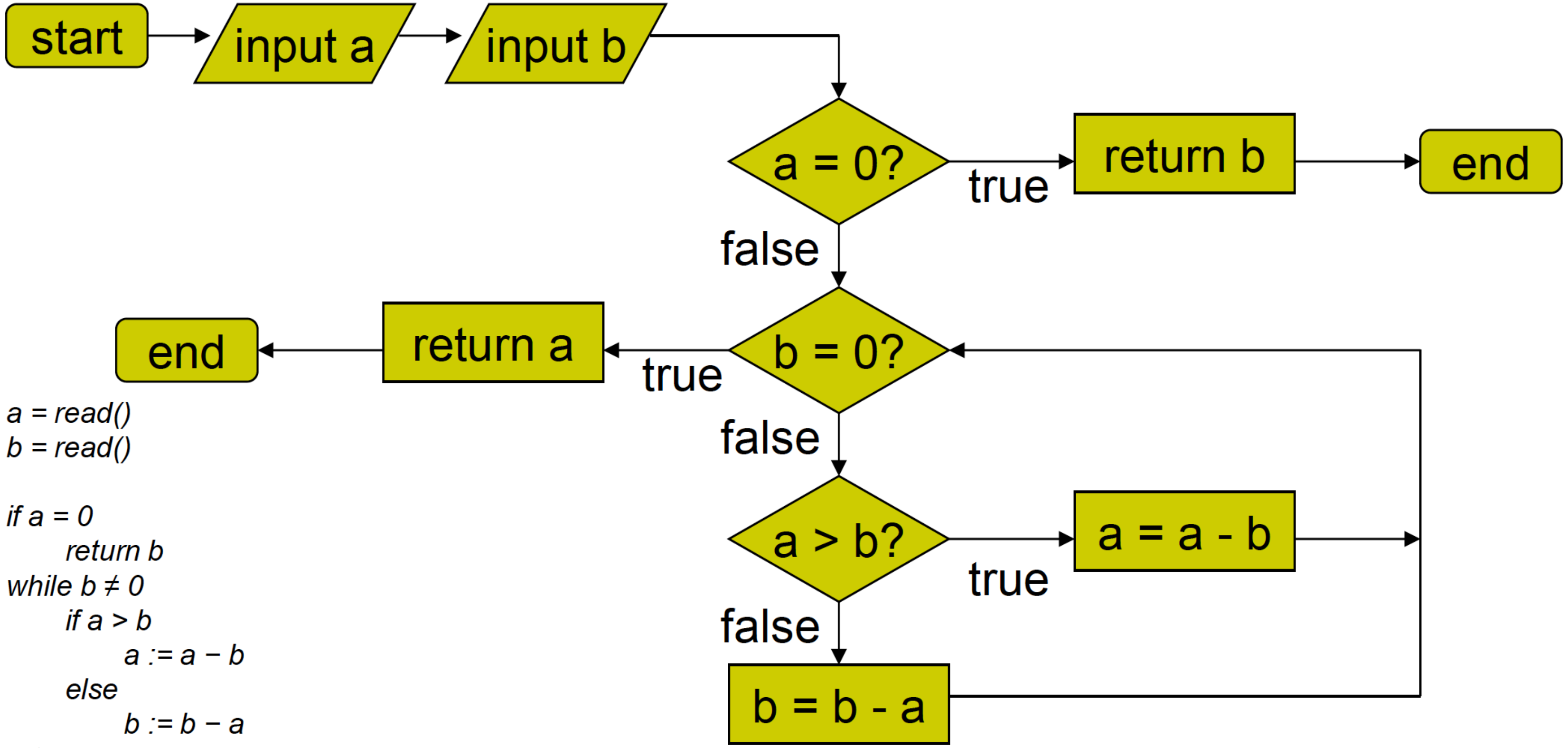
Flowcharts

- Graphical representation of the behavior of an algorithm
  - Represents the steps of an algorithm by geometric shapes
  - Temporal relationships are shown by connections

- Developed in the early 20th century for use in industrial engineering
  - Used in many domains for the last 100 years
  - John von Neumann developed the flow chart while working at IBM as a means to describe how programs operated
  - Flowcharts are still used to describe computer algorithms- UML activity diagrams are an extension of the flowchart

- There are many flowchart notation standards

Flowcharts




```
a = read()
b = read()

if a = 0
    return b
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

Flowcharts

Strengths and weaknesses

- The set of defined constructs is both minimal and complete
- The resulting algorithms can be hard to understand and analyze
- Graphical methods do not scale well – very difficult to represent large or complex algorithms
- Hard to distribute, share, and reuse

Finite State Machines (FSM)

- Behavioral models composed of a finite number of states, transitions between those states, and actions

- FSMs are represented by state diagrams

- State diagrams have been used for 50+ years in software, hardware, and system design and there are a variety of notations and approaches

  – Traditional Mealy-Moore state machines
  – Harel state machines
  – UML state machines

Finite State Machines (FSM)

A traditional (e g. Mealy-Moore) type of FSM is a quintuple $(\Sigma, S, s_0, \delta, F)$

- Σ is the input alphabet where Σ is finite $\wedge$ $\Sigma \neq \emptyset$
- S is a set of states where S is finite $\wedge$ $S \neq \emptyset$
- s_0 is an initial state where, $s_0 \in S$
- $\delta(q, x)$ is the state transition function where $q \in S \wedge x \in \Sigma$
  - (If the FSM is nondeterministic, then δ could be a set of states)
- F is the set of final states where $F \subseteq$ of $S \cup \{\emptyset\}$

Finite State Machines (FSM)

$\delta(q, x)$ may be a partial function:

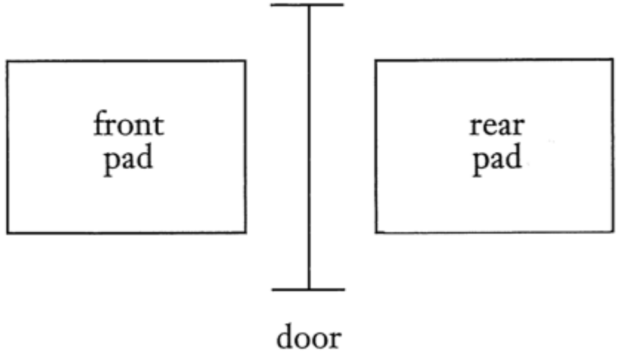
- $\delta(q, x)$ does not have to be defined for every combination of q and x
- If it is not defined, then the FSM can enter an error state or reject the input

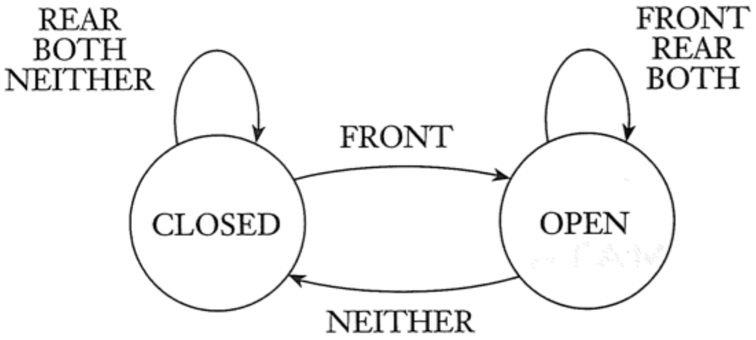
Finite State Machines (FSM)

The following (limiting) assumptions are made regarding traditional (deterministic) Mealy-Moore FSAs

- an FSA can only be in one state at a time, and must be in exactly one state at all times
- States of one FSA are independent from the states of all other FSAs
- Transitions between states are not interruptible
- Actions are atomic and run to completion
- Actions may be executed on entry into a state, on exit from a state, or during the transition from one state to another

Finite State Machines (FSM)





input signal

state	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Finite State Machines (FSM)

Transitions indicate state change from one state to another that are described by

- a condition that needs to be fulfilled to enable a transition
- an action which is an activity that is to be performed at some point in the transition
    - Entry action: which is performed when entering the state
    - Exit action: which is performed when exiting the state
    - Input action: which is performed depending on present state and input conditions
    - Transition action: which is performed when performing a certain transition

Finite State Machines (FSM)

- Popular form of FSM are the Harel State Diagrams

- A variant which was adopted for Unified Modeling Language (UML) State Machines

- There are two types of UML State Machines
  - Behavioral State Machines (BSM)
    Model the behaviour of objects

  - Protocol State Machines (PSM)
    Model protocols of interfaces and ports

- Most use users of UML don't differentiate

Finite State Machines (FSM)

FSAs are limited and it is difficult to model concurrency, complex object states, threads, multi-tasking

UML state machines extend the traditional automata theory in several ways that include

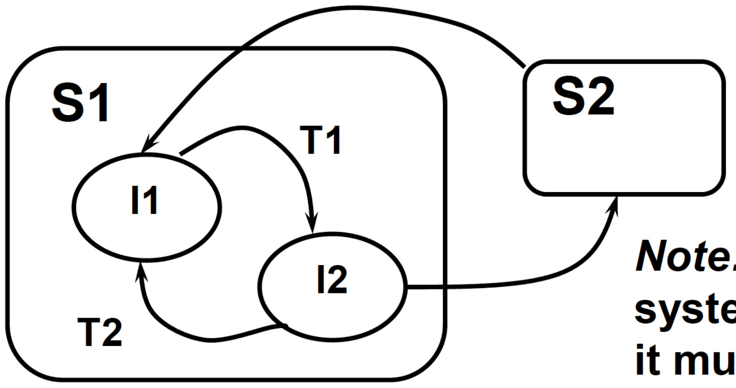
- nested state
- guards
- actions
- activities
- orthogonal components
- concurrent state models

Finite State Machines (FSM)

UML State Machines – Nested States

Outer state is called the superstate

Inner states are called substates

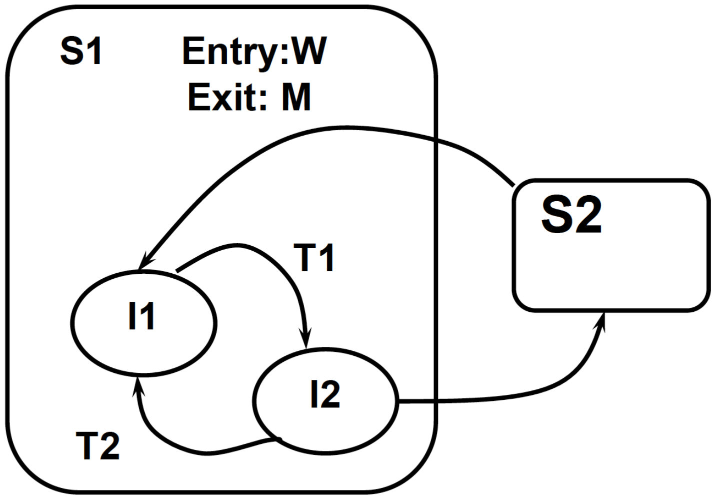


Note: **while the system is in state S1 it must be in exactly one state**

Finite State Machines (FSM)

UML State Machines – Actions

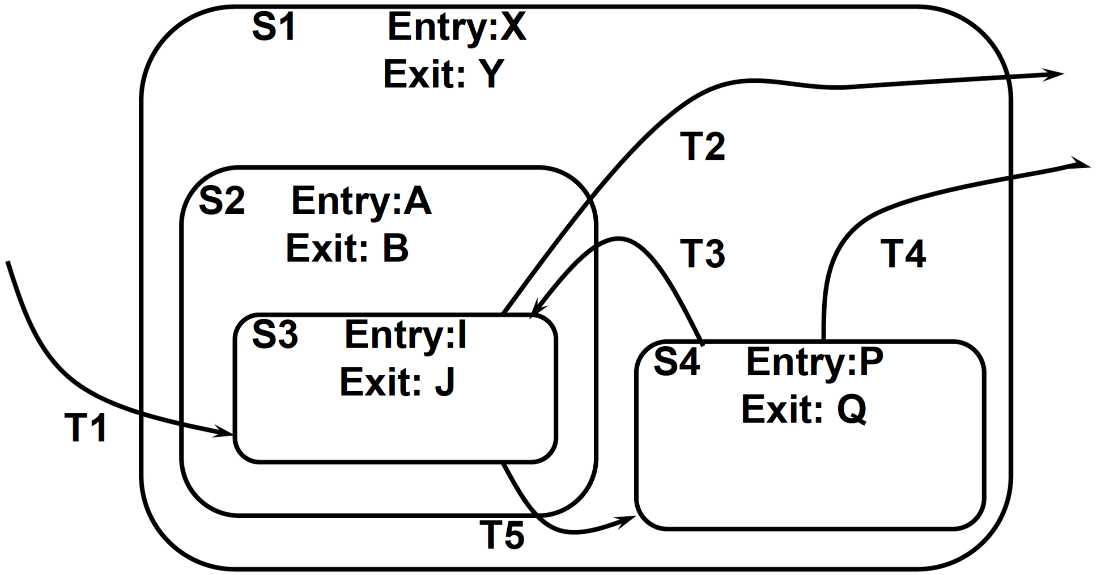
You can specify state entry and exit actions



Finite State Machines (FSM)

UML State Machines – Actions

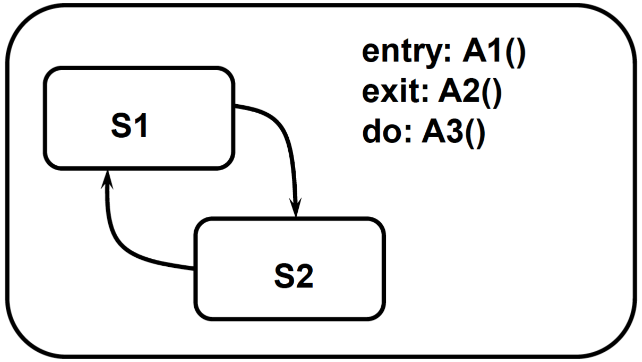
You can nest entry and exit actions



Finite State Machines (FSM)

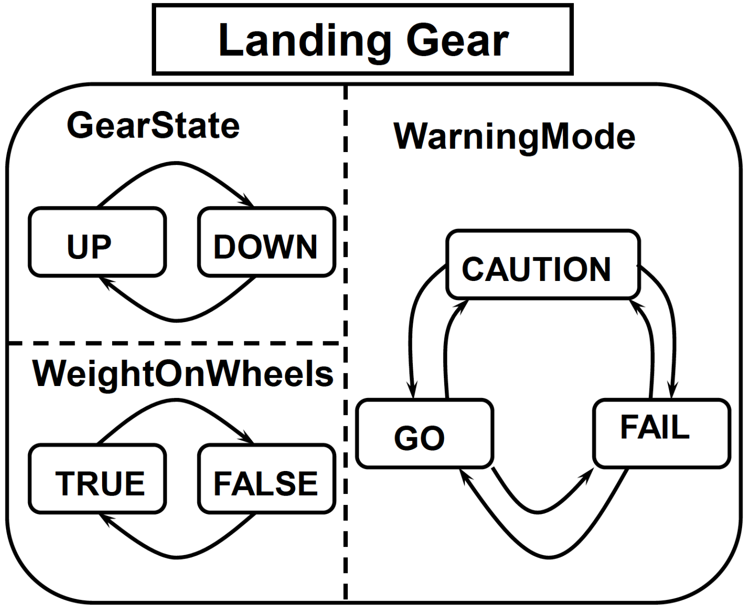
UML State Machines – Activities

- Like actions except they are performed as long as the state is active
- Activities are indicated with a do: statement



Finite State Machines (FSM)

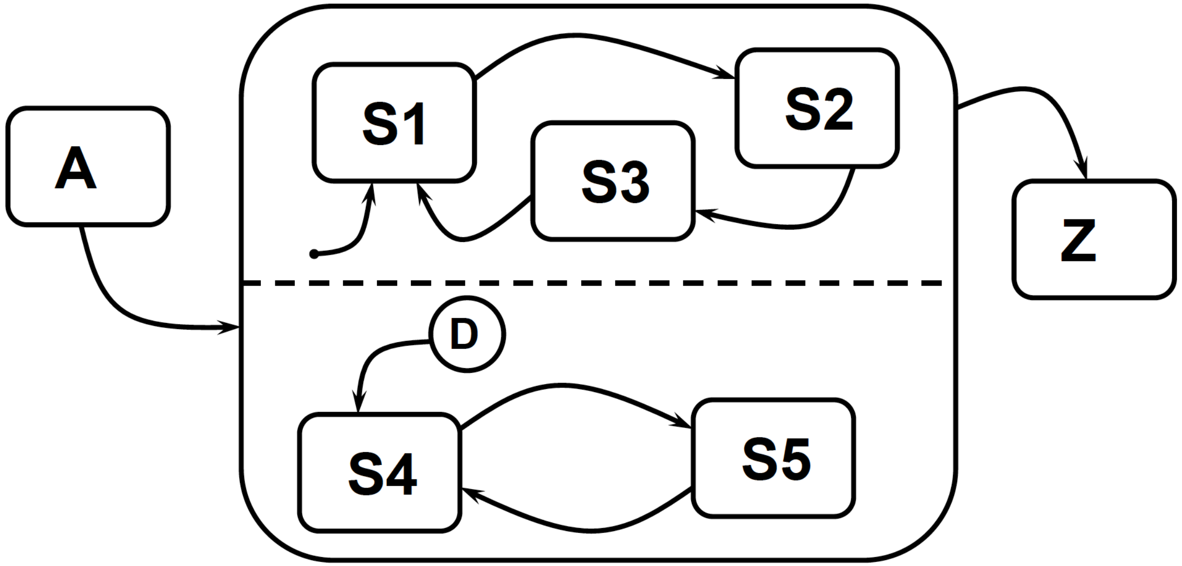
UML State Machines – Orthogonal Components



Finite State Machines (FSM)

UML State Machines – Concurrent State Models

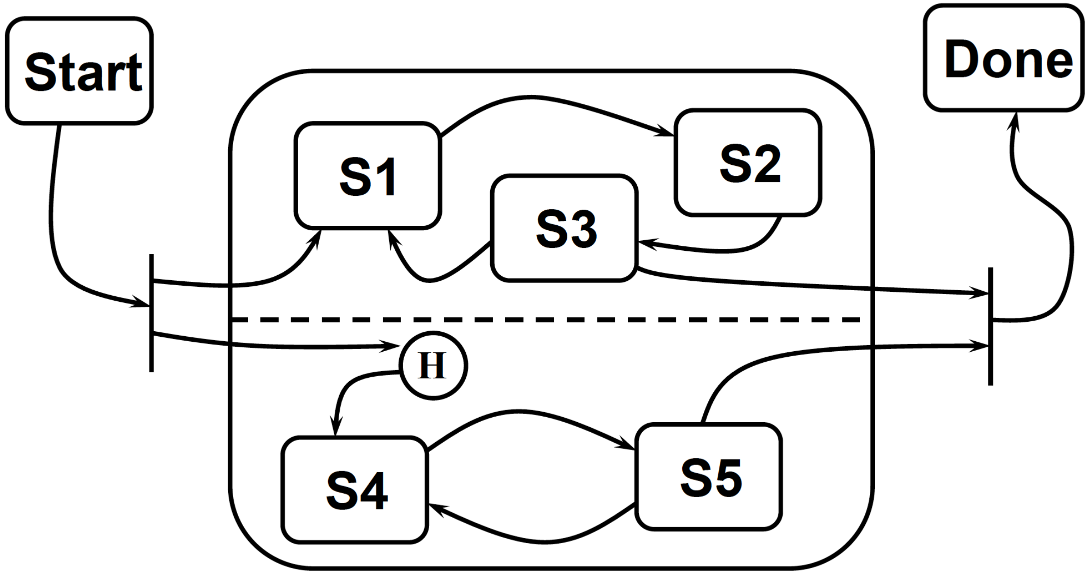
Concurrent threading can be modelled



Finite State Machines (FSM)

UML State Machines – Concurrent State Models

Forking / Joining can be modelled



Finite State Machines (FSM)

- In traditional FSA, transitions carry little or no information
- UML state machine transitions carry a lot of information:
  - Event Name - Name of triggering event
  - Parameters - data passed with event
  - Guard - condition that must be true for the transition to occur
  - Action List - list of actions executed
  - Event List - list of events executed

Finite State Machines (FSM)

- The key problem with FSM technologies is that they simply do not scale up well
  - State explosion is a common problem
  - Care must be taken to restrict the scope of what is being modeled
- FSMs often abstract away the very algorithms we want to model
  - Care must be taken to maintain a proper and consistent level of abstraction
  - Can violate notions of completeness

Finite State Machines (FSM)

UML state machines are really more like a notation than traditional FSMs

- Violates minimalism
- Any benefit gain in applying mathematical rigor may be lost