04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 1

Introduction & The Software Development Life Cycle

- Motivation
- Goals of the course
- Syllabus & lecture schedule
- Course operation
- Preview of selected course material
- Software development tools for exercises and assignments
- Exercises
- Levels of abstraction in information processing systems
- The software development life cycle
  - Yourdon Structured Analysis
- Software process models
  - Waterfall, Evolutionary, Formal Transformation, Re-Use, Hybrid, Spiral

Motivation

Software is everywhere, not only in IT sectors:

- Robotics & automation
- Automotive
- Aerospace
- Communications
- Medical
- Energy distribution and management
- Environmental control
- ...

Motivation

Most software is in embedded systems

- Highly constrained in terms of
  - Memory
  - Processing power
  - Bandwidth
- Have exacting requirements for reliability, safety, availability

Motivation

Engineers who develop the software

- Do not always have a strong background in
  - Computer science
  - Computer engineering
  - Algorithms
  - Data Structures
- Formal education in other engineering disciplines

Motivation

This is a problem …

– Suppose you've developed a software application
– And it works just fine in the current set of circumstances
– But can you be sure it will scale?
    - Larger data sets (input)
    - Larger user base
    - Tighter time and memory constraints
    - Migration to a distributed computing environment
– This is where a solid foundation in data structures & algorithms comes in

Motivation

Example 1

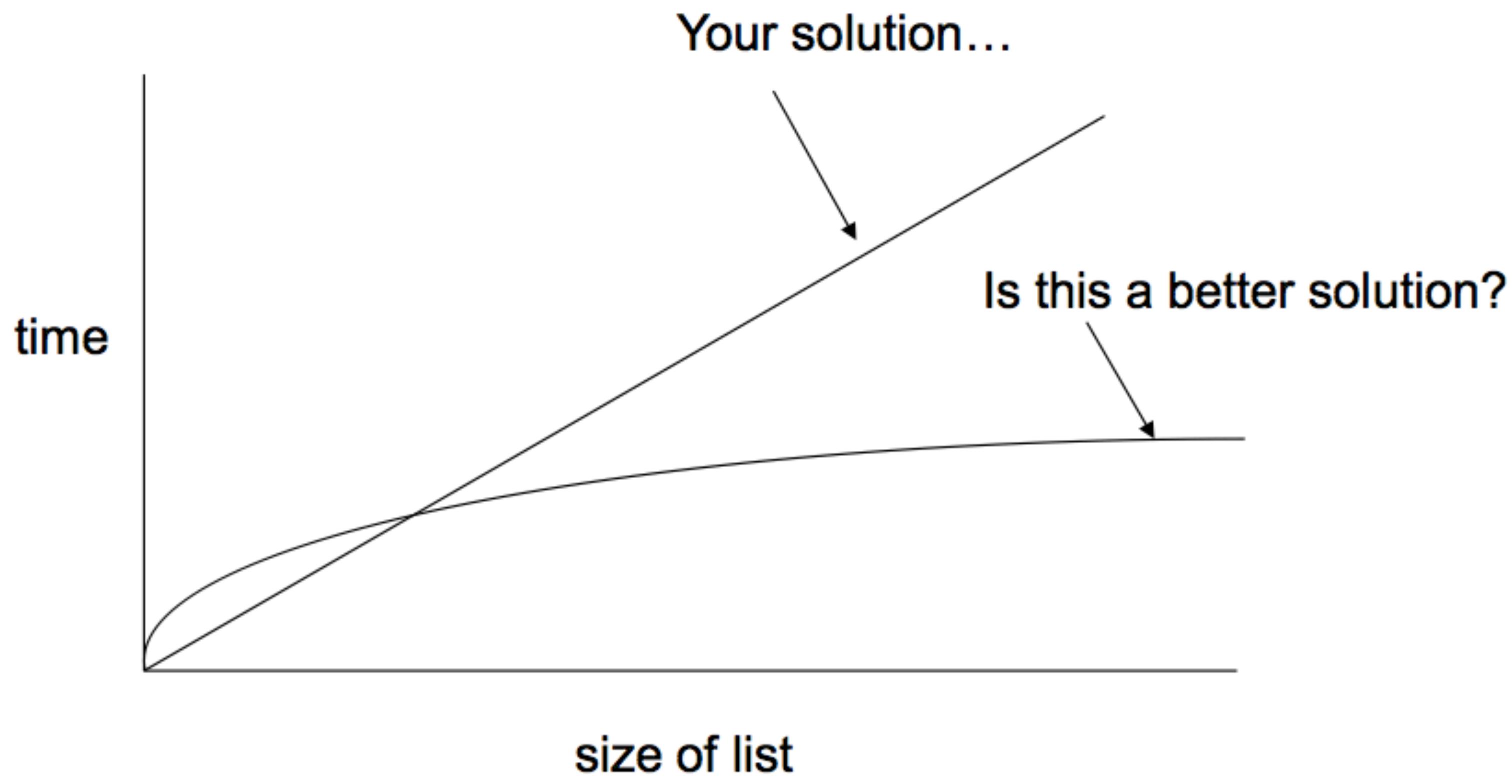
- **Problem**: Your program needs to find whether a list stored in memory contains a particular data element

- **Your solution**: Start from the beginning of the list and examine each element

- How good is this? What does it depend on?
- Can you do better?
- Under what circumstances could you improve this?
- Is the list the optimal data structure for this?

Motivation

Example 1



Motivation

Example 1

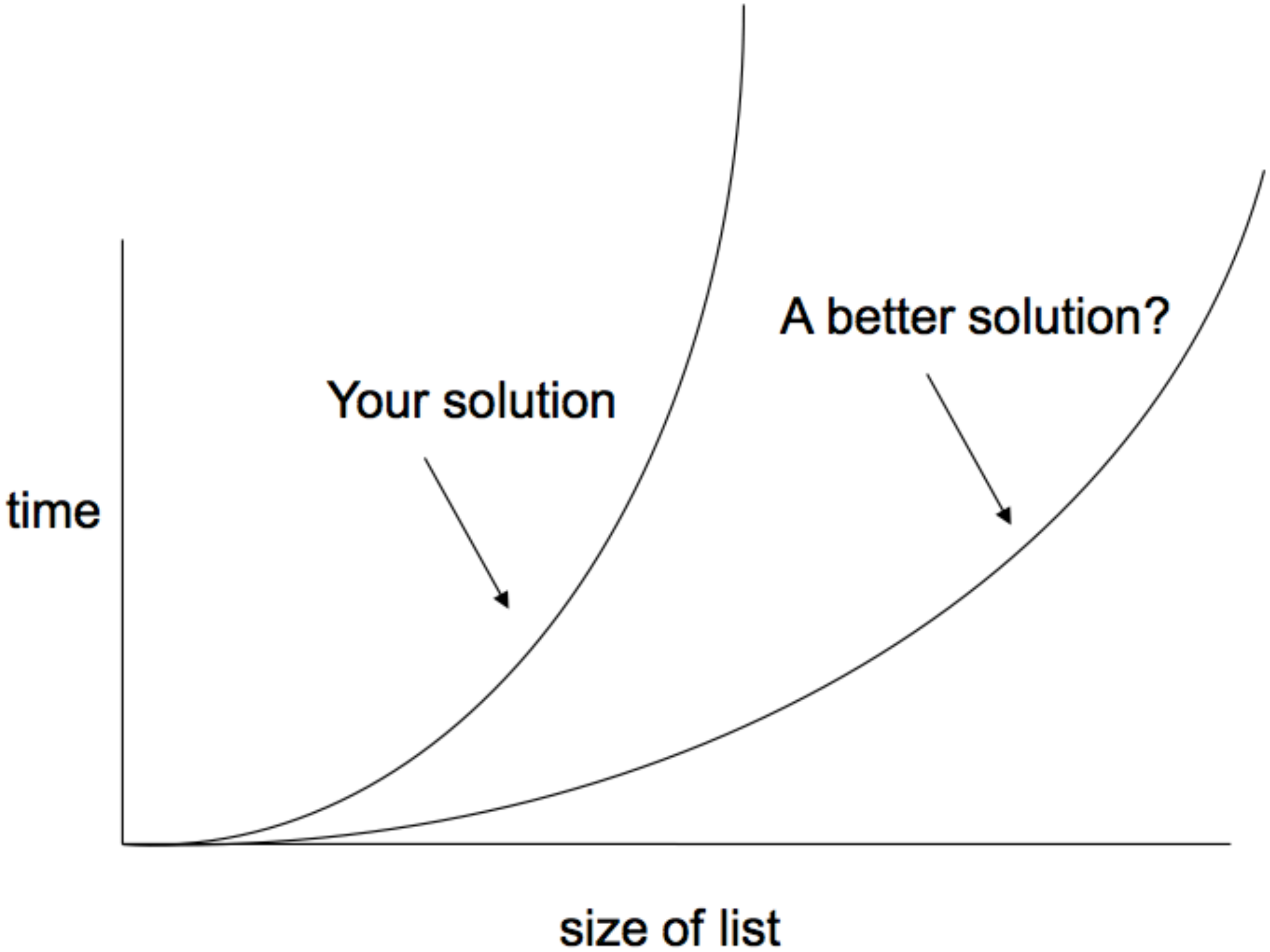
- **Problem:** You need to output a sorted list of elements stored in memory

- **Your solution:** Find and output the largest; find and output the next largest, ...

- How good is this?
- Can you do better?
- Under what circumstances?

Motivation

Example 2



Motivation

Example 3

- **Problem:** You are creating a car navigation assistant to devise a route that will allow the driver to visit a set of cities optimally, e.g., minimize fuel consumption, distance, or time

- This is the classic Travelling Salesman problem

- **Your solution:** List out all possible ways of visiting all cities. Select the one that minimizes the total distance traveled

- How good is this?
- Can you do better?

Motivation

Example 3

– **Your solution:**
  Assuming 1 microsecond to generate each path:

Cities	Computing time
2	Really fast
7	~1 Second
11	~1 Hour
12	~1 Day
14	~1 Year
17	~1 Century

– Can you do better? If so, what will it take?

Motivation

Example 4

- **Problem:** You have an system with a lot of legacy code in it, much of it is believed to be obsolete. You want to write a general program to find the code segments that are never actually executed in a system, so that you can then remove them

- Your solution: ?????

Motivation

So What?

- We have seen instances of four kinds of problem complexity that occur all the time in industry
  - Linear
  - Polynomial
  - Exponential
  - Undecidable

- Knowing which category your problem fits into is crucial
  - You can use special techniques to improve your solution

Motivation

So What?

- Competitive advantage is based on the characteristics of products sold or services provided
  - Functionality, timeliness, cost, availability, reliability, interoperability, flexibility, simplicity of use
- Innovation will be delivered through quality software
  - 90% of the innovation in a modern car is software-based
- Software determines the success of products and services

Goals of the Course

- Provide engineers who don't have a formal background in computer science with a solid foundation in the key principles of data structures and algorithms

- Leverage what software development experience they do have to make them more effective in developing efficient software-intensive systems

Goals of the Course

- Foster algorithmic thinking
- Appreciate the link between
  - Computational theory
  - Algorithms and Data Structures
  - Software implementation
- Impart professional practical skills in software development
- Develop the ability to recognize & analyze critical computational problems and assess different approaches to their solution

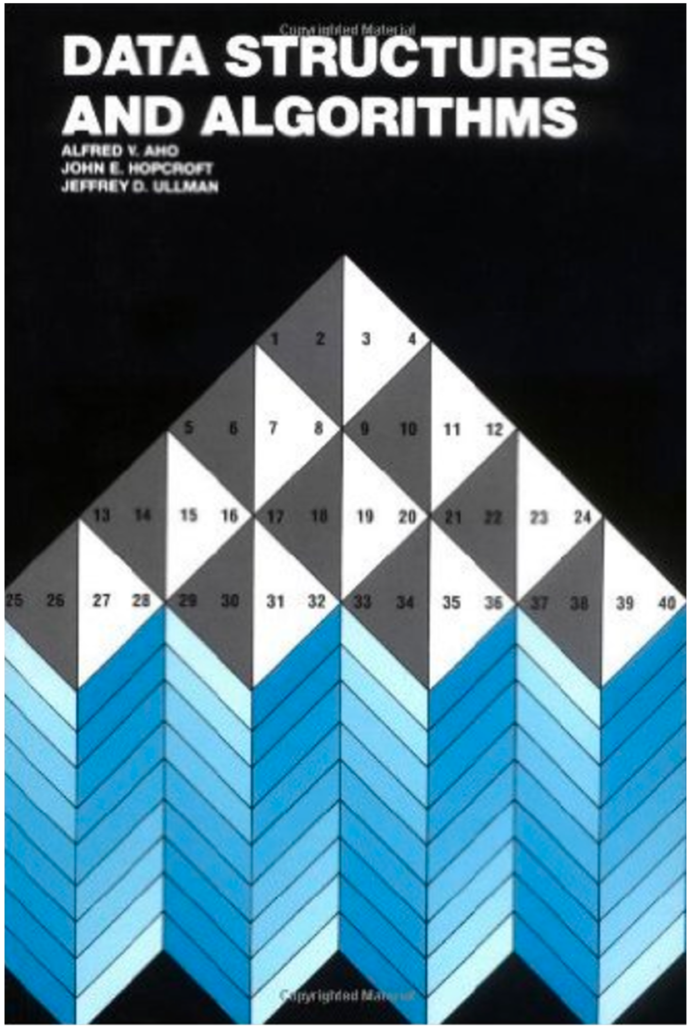
Goals of the Course

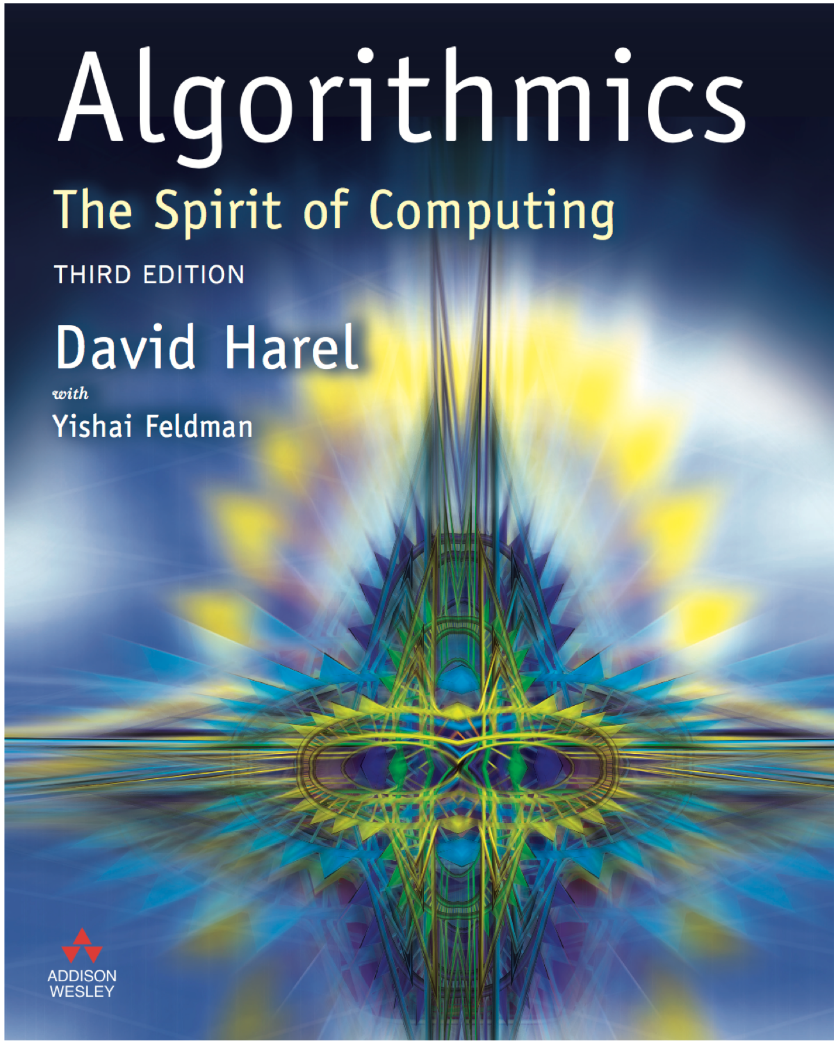
Key themes

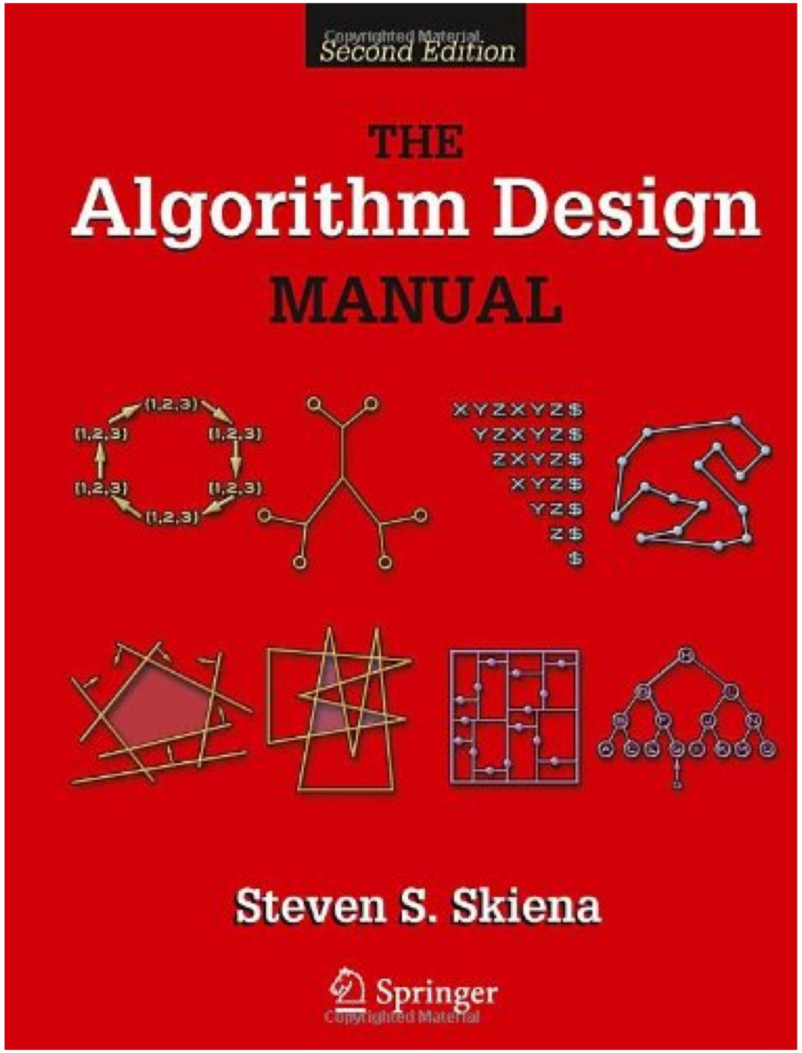
- Principles and practice (analysis and synthesis)
- Practical hands-on learning (lots of examples)
- Detailed implementation, not just pseudo-code
- Broad coverage of the essential tools in algorithms and data structures

Syllabus & Lecture Schedule

https://canvas.cmu.edu/courses/3210


DATA STRUCTURES AND ALGORITHMS
ALFRED V. AHO
JOHN E. HOPCROFT
JEFFREY D. ULLMAN


Algorithmics
The Spirit of Computing
THIRD EDITION
David Harel
with
Yishai Feldman
ADDISON WESLEY


Copyrighted Material
Second Edition
THE
Algorithm Design
MANUAL
Steven S. Skiena
Springer
Copyrighted Material

Course Operation

Course Operation

- Lectures will be posted in advance: read them before coming to class and read them again after class
- Readings: read them after class
- Assignments & Assessment
  - 7 individual programming assignments (10% each; best six)
  - Mid-semester examination (10%)
  - Final examination (30%)
  - Marking schemes will be distributed in due course
    - Functionality (based on testing using an unseen data set)
    - Documentation: internal and external
    - Tests and testing strategy
  - Strict deadlines: NO EXTENSIONS except on compassionate grounds

Course Operation

We will have a 10 minute quiz every Friday to kick off recitation

- Style will vary:
  - Some will be multiple choice (negative marking will apply)
  - Some will match that of one section of a question in the final examination
- Not for credit
- Not an assessment exercise
- Learning exercise
  - We will work through the solution together during the recitation hour and use it to prompt questions

Course Operation

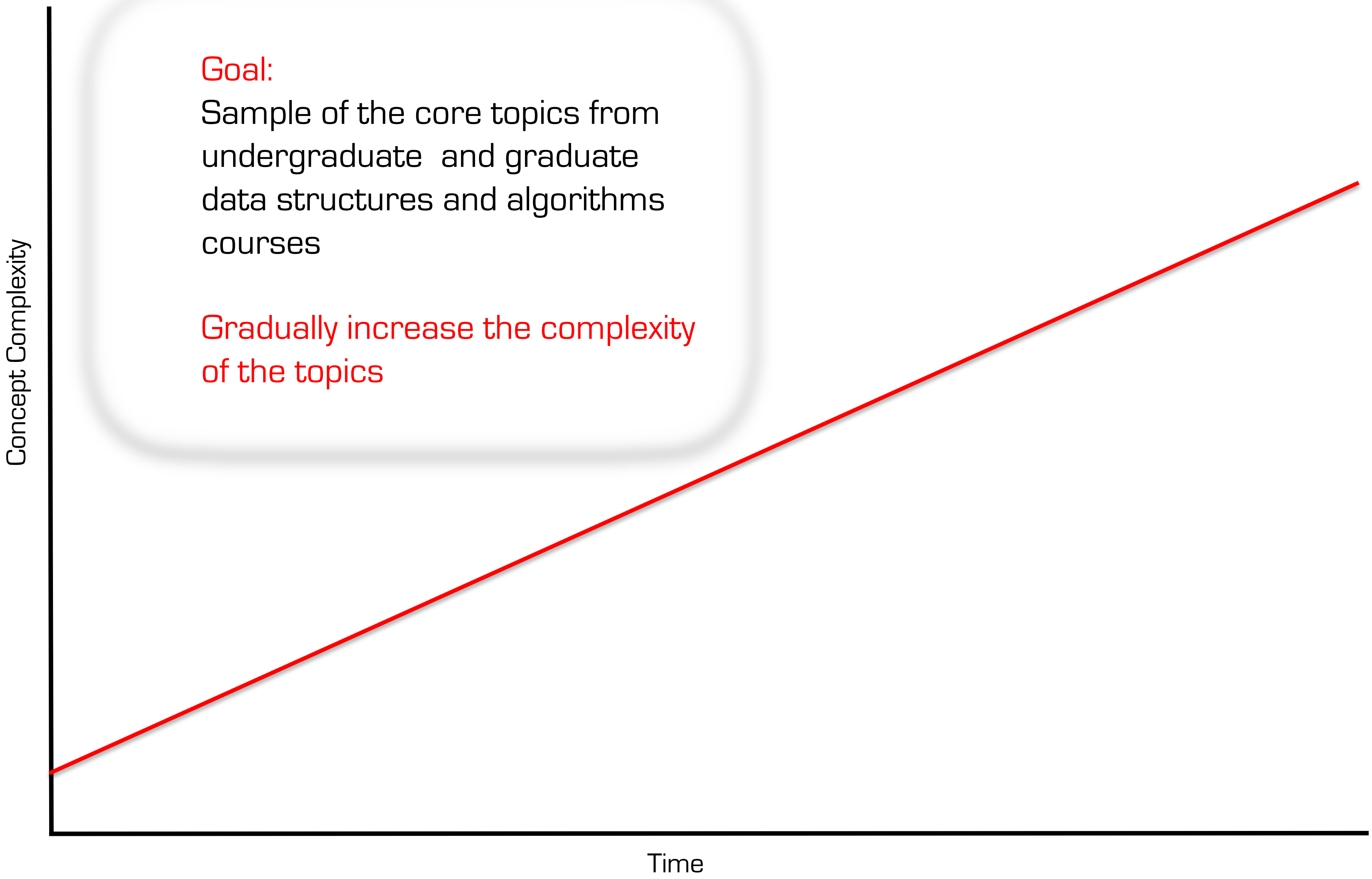
- Do
  - Participate in class
  - Ask questions (you will be doing others a favour)
  - Discuss course material, readings, assignments with other students
  - Share thoughts but not written material (e.g. code, documentation)
  - Cite any work you use in assignments
  - Be a good teammate: do your fair share of the work equally & cooperate

- Don't
  - Cheat or plagiarize
    - Uncited use of any material from anywhere
    - Share / steal any material with/from former or current students

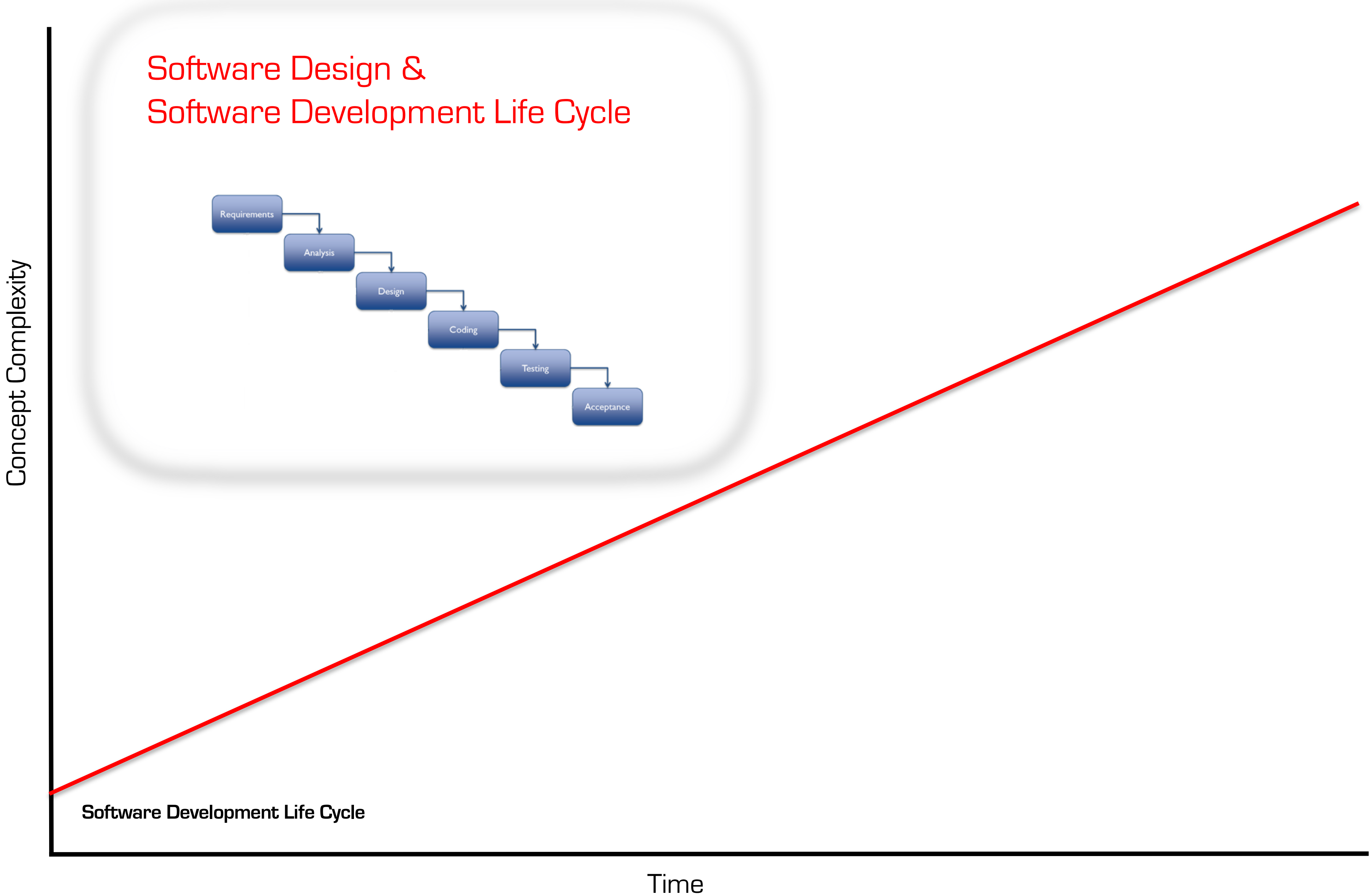
- Sanctions for cheating and plagiarism
  - Zero marks for first sharing infingement (both parties)
  - Fail the course (grade R) for second sharing infringement (both parties)
  - Fail the course (grade R) for first stealing infringement

Preview of Selected Course Material

Data-Structures and Algorithms for Engineers



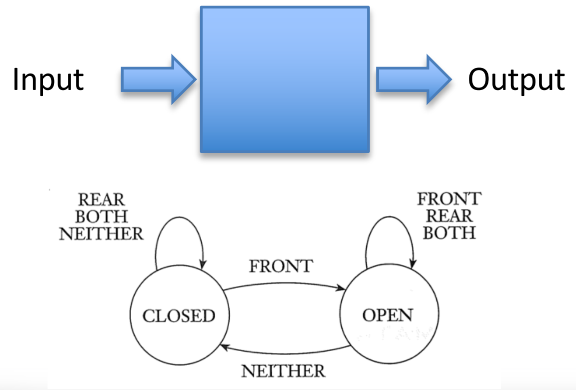
Data-Structures and Algorithms for Engineers



Data-Structures and Algorithms for Engineers

Formalisms for representing algorithms

I/O, Flow-charts, Pseudo-code, FSM, UML, ….



Concept Complexity

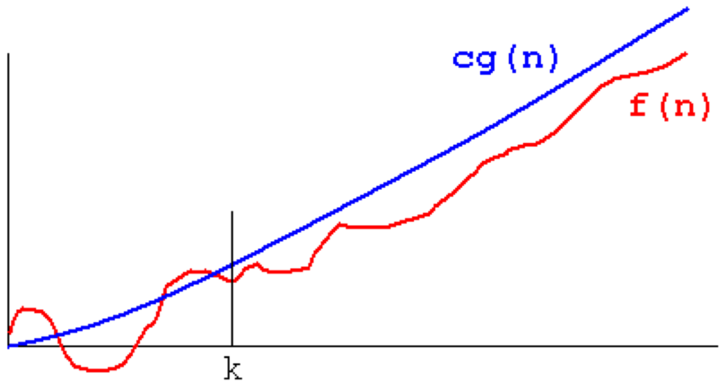
Formalisms representing algorithms

Software Development Life Cycle

Time

Data-Structures and Algorithms for Engineers

Analysis of Complexity



- Big O notation
- Recurrence relationships
- Analysis of complexity
- Iterative and recursive algorithms

Analysis of Complexity

Formalisms representing algorithms

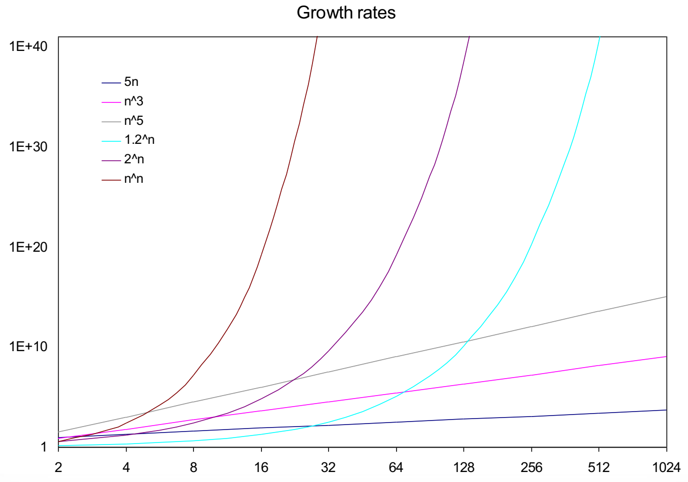
Software Development Life Cycle

Concept Complexity

Time

Data-Structures and Algorithms for Engineers

Analysis of Complexity



- Tractable, intractable complexity
- Determinism and non-determinism
- P, NP, and NP-Complete classes of algorithm

Concept Complexity

Analysis of Complexity

Formalisms representing algorithms

Software Development Life Cycle

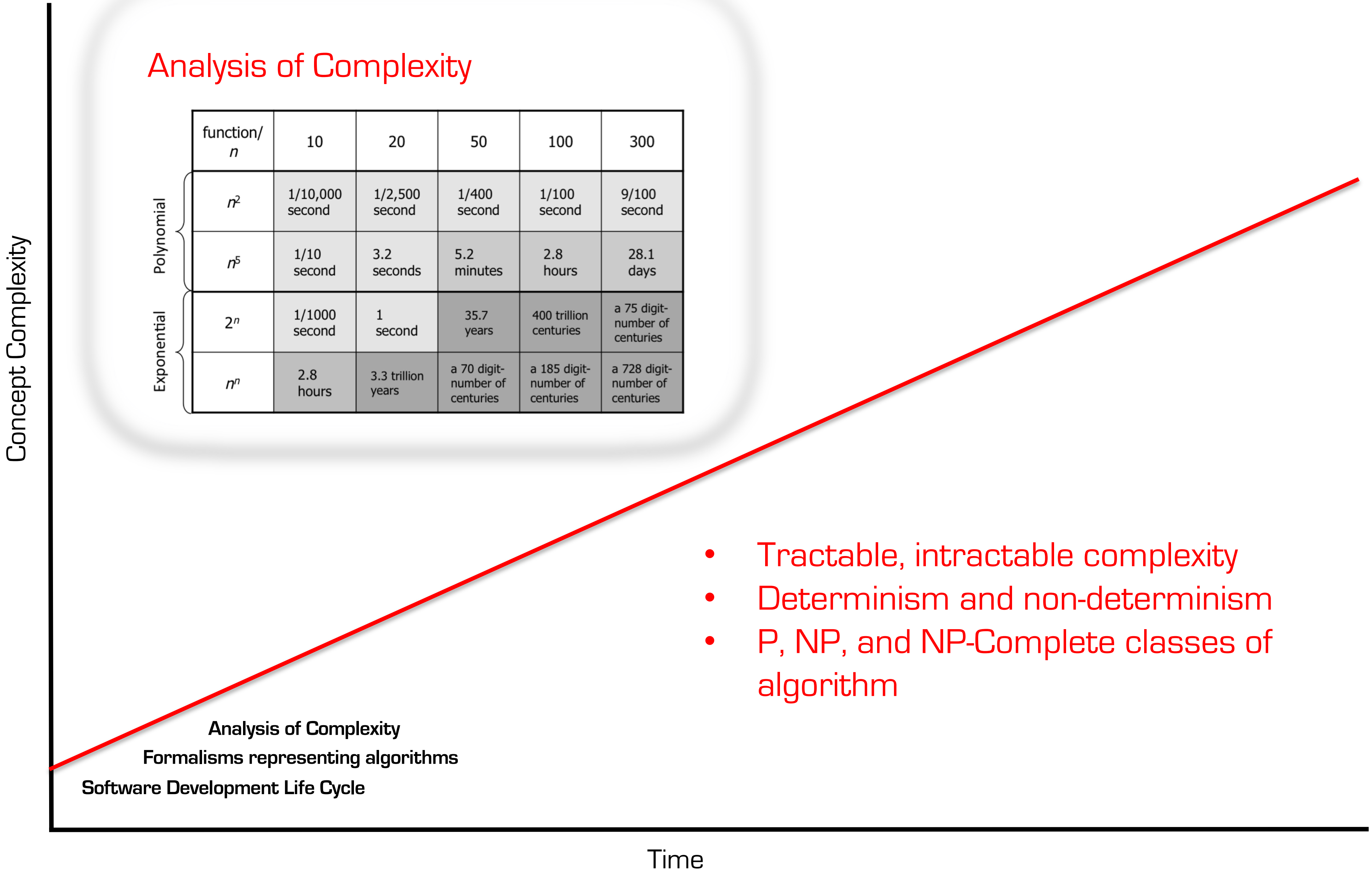
Time

Data-Structures and Algorithms for Engineers

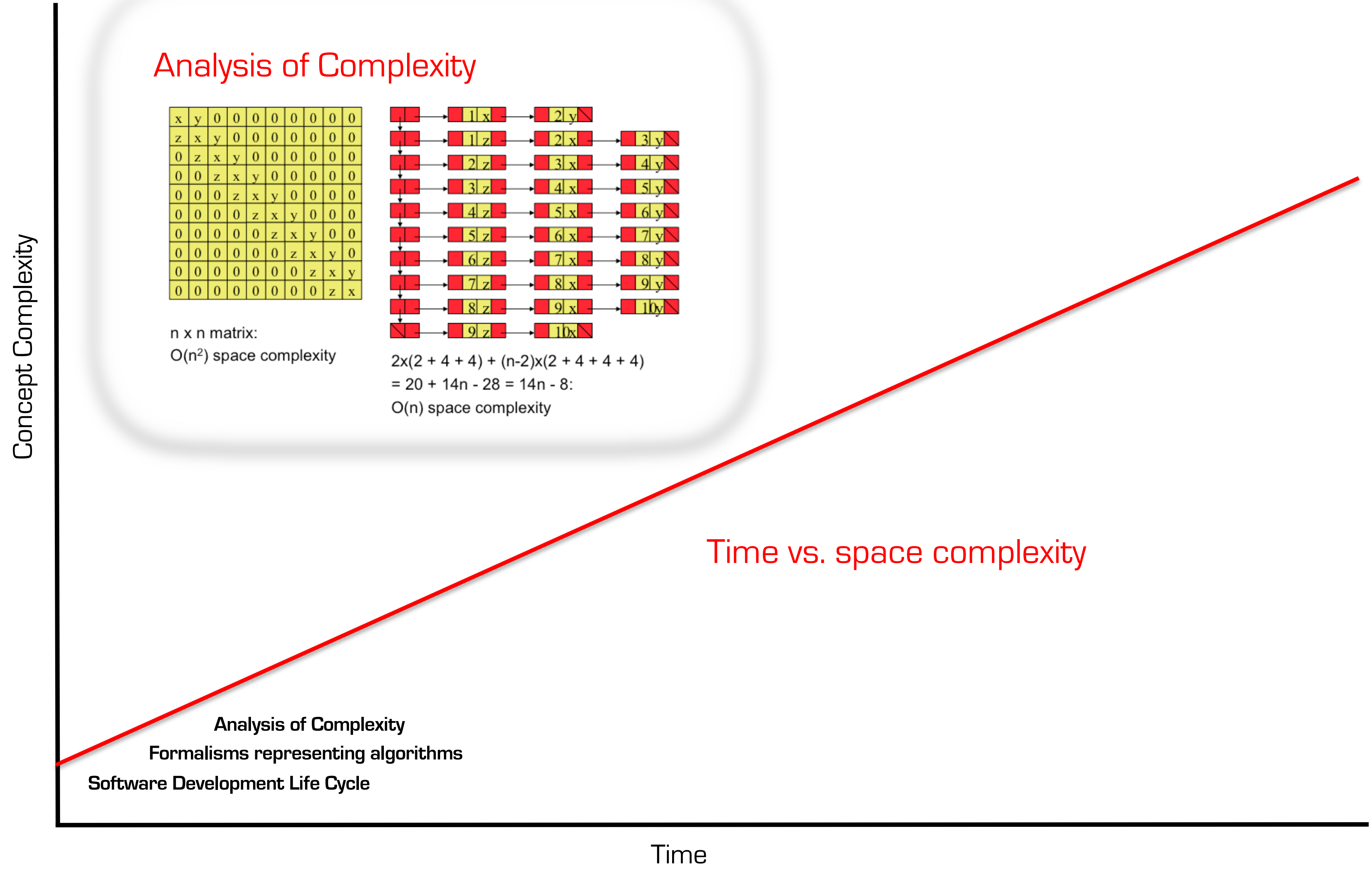
Analysis of Complexity

	function/ n	10	20	50	100	300
Polynomial	n^2	1/10,000 second	1/2,500 second	1/400 second	1/100 second	9/100 second
	n^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
Exponential	2^n	1/1000 second	1 second	35.7 years	400 trillion centuries	a 75 digit-number of centuries
	n^n	2.8 hours	3.3 trillion years	a 70 digit-number of centuries	a 185 digit-number of centuries	a 728 digit-number of centuries

- Tractable, intractable complexity
- Determinism and non-determinism
- P, NP, and NP-Complete classes of algorithm



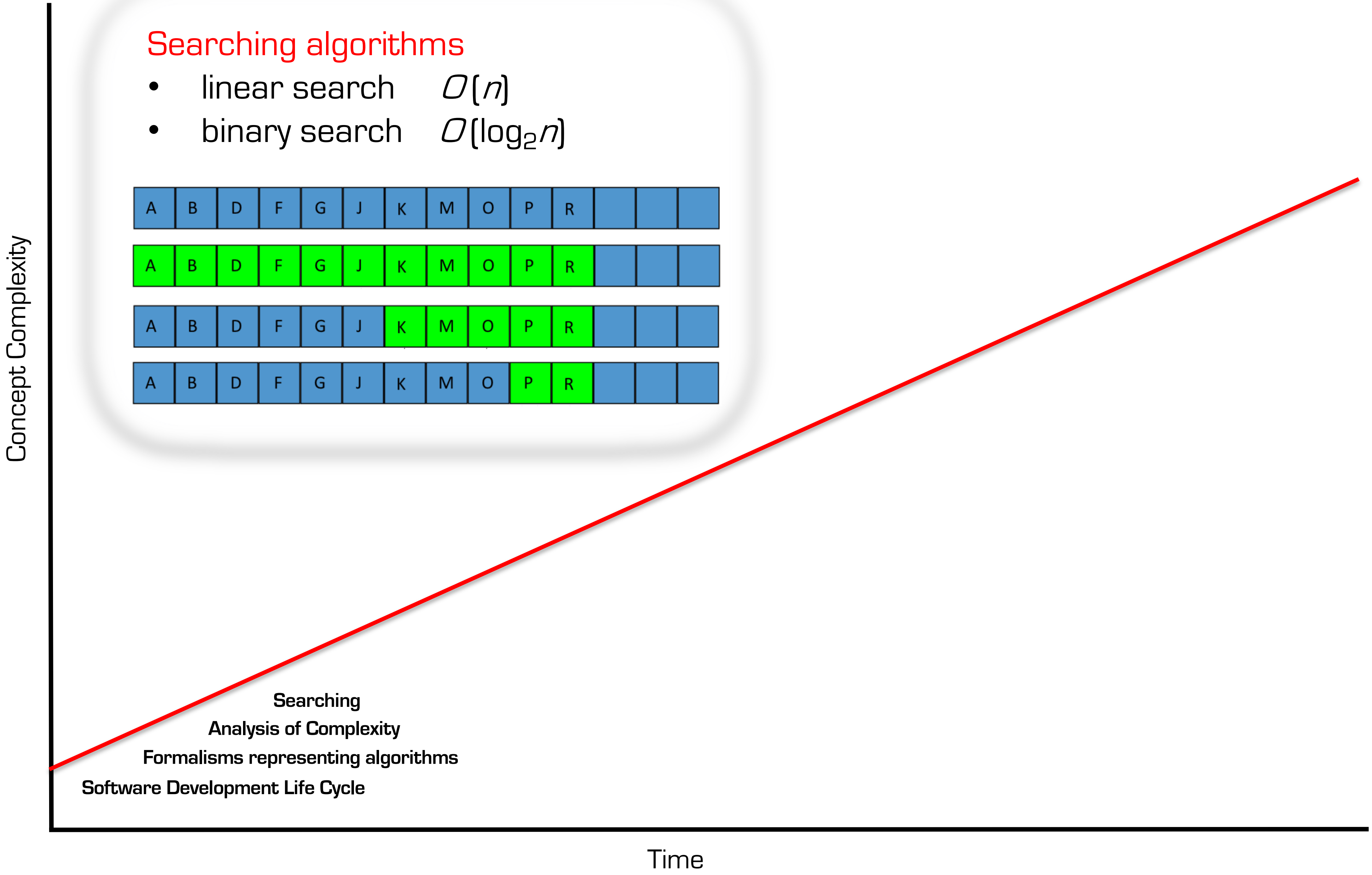
Data-Structures and Algorithms for Engineers



Data-Structures and Algorithms for Engineers

Searching algorithms

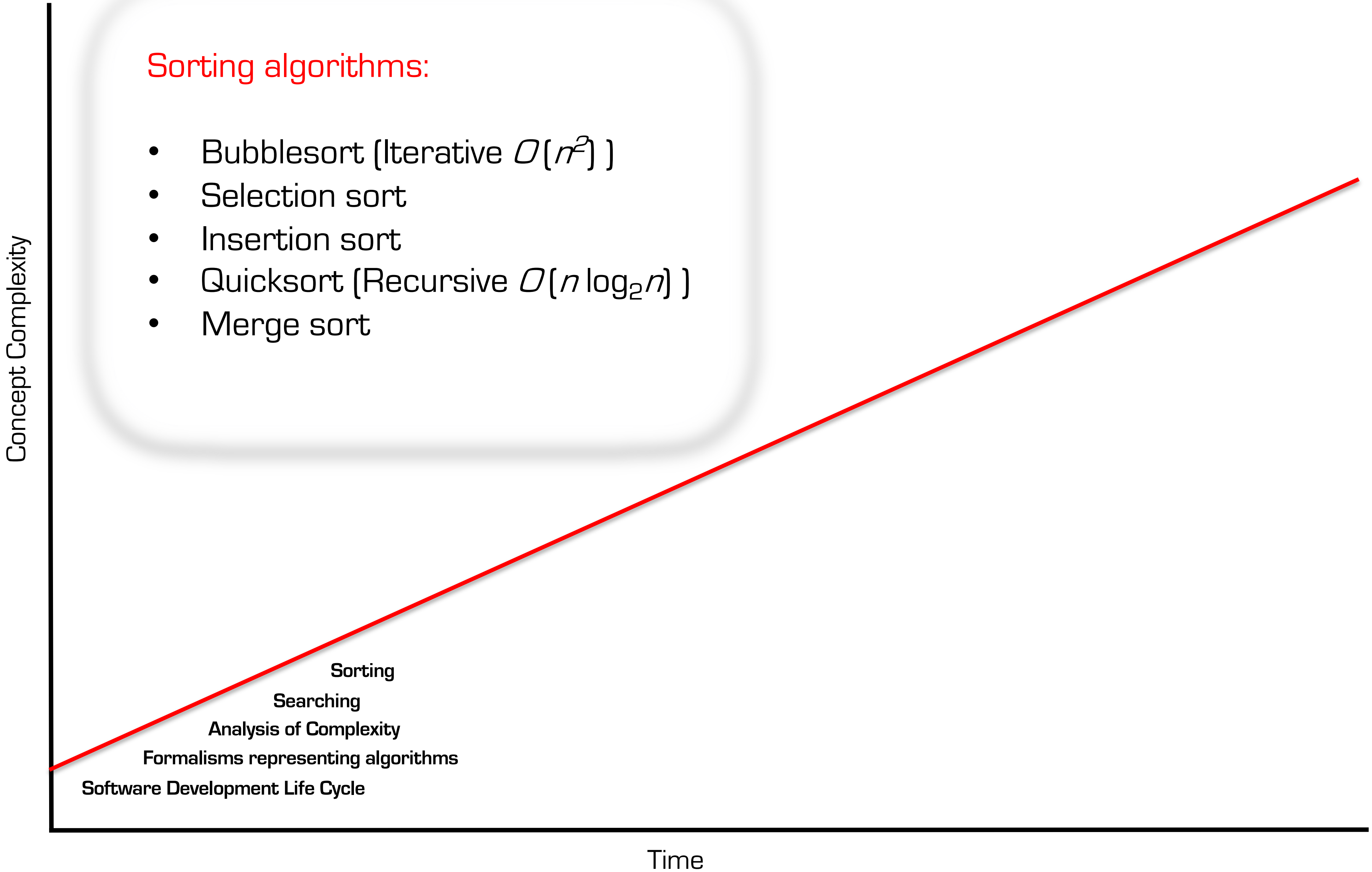
- linear search $O(n)$
- binary search $O(\log_2 n)$



Data-Structures and Algorithms for Engineers

Sorting algorithms:

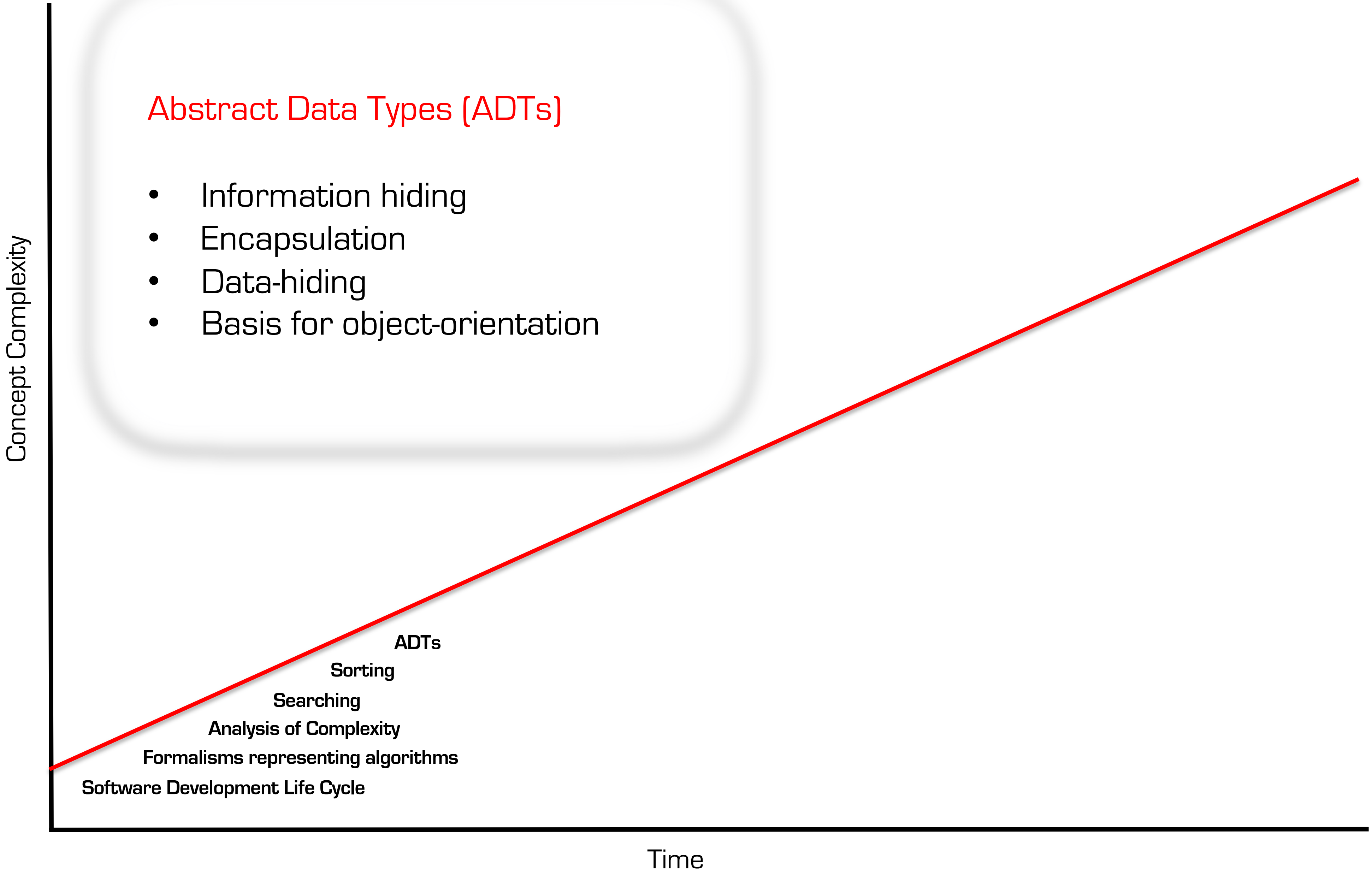
- Bubblesort (Iterative $O(n^2)$)
- Selection sort
- Insertion sort
- Quicksort (Recursive $O(n \log_2 n)$)
- Merge sort



Data-Structures and Algorithms for Engineers

Abstract Data Types (ADTs)

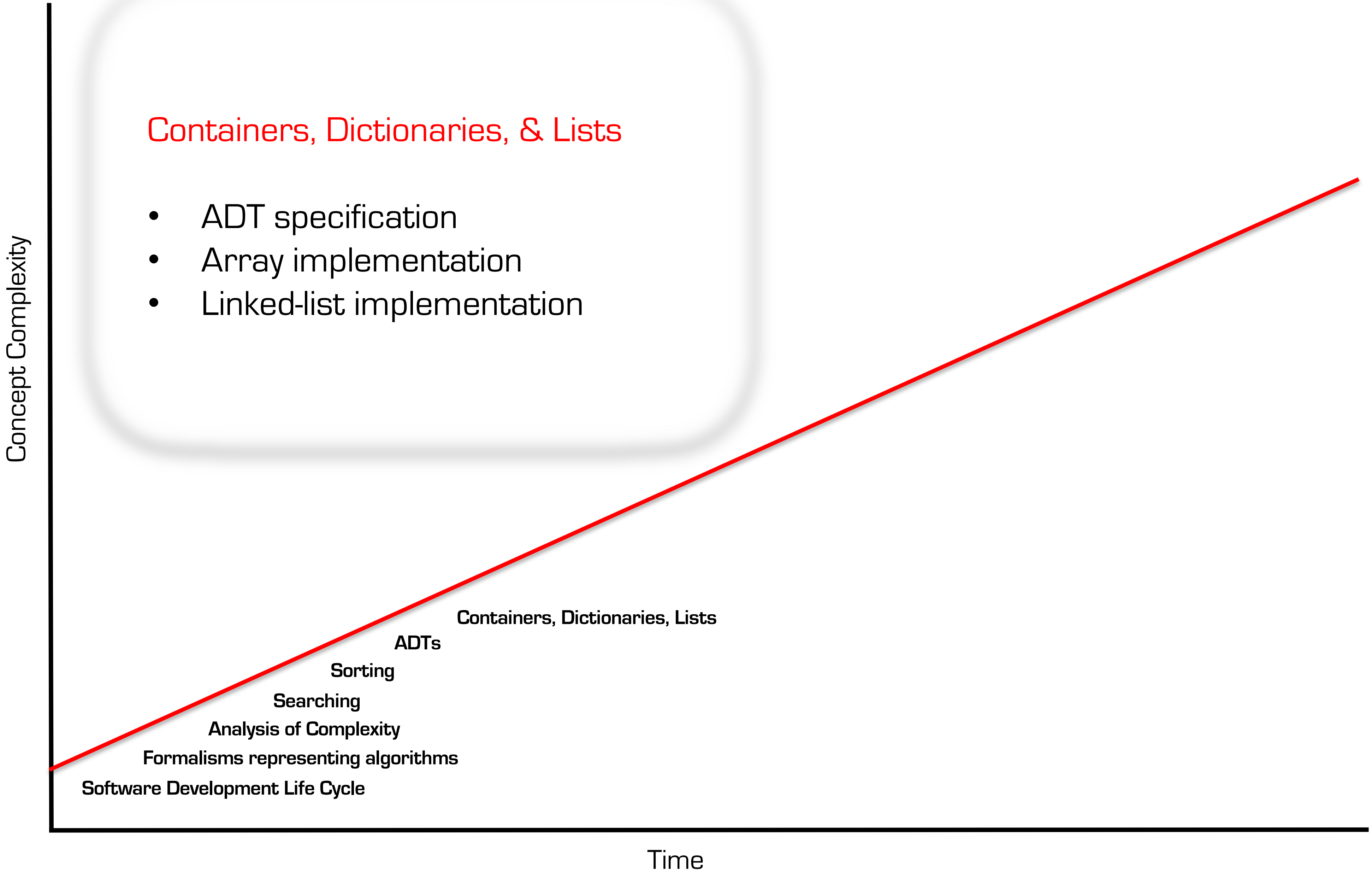
- Information hiding
- Encapsulation
- Data-hiding
- Basis for object-orientation



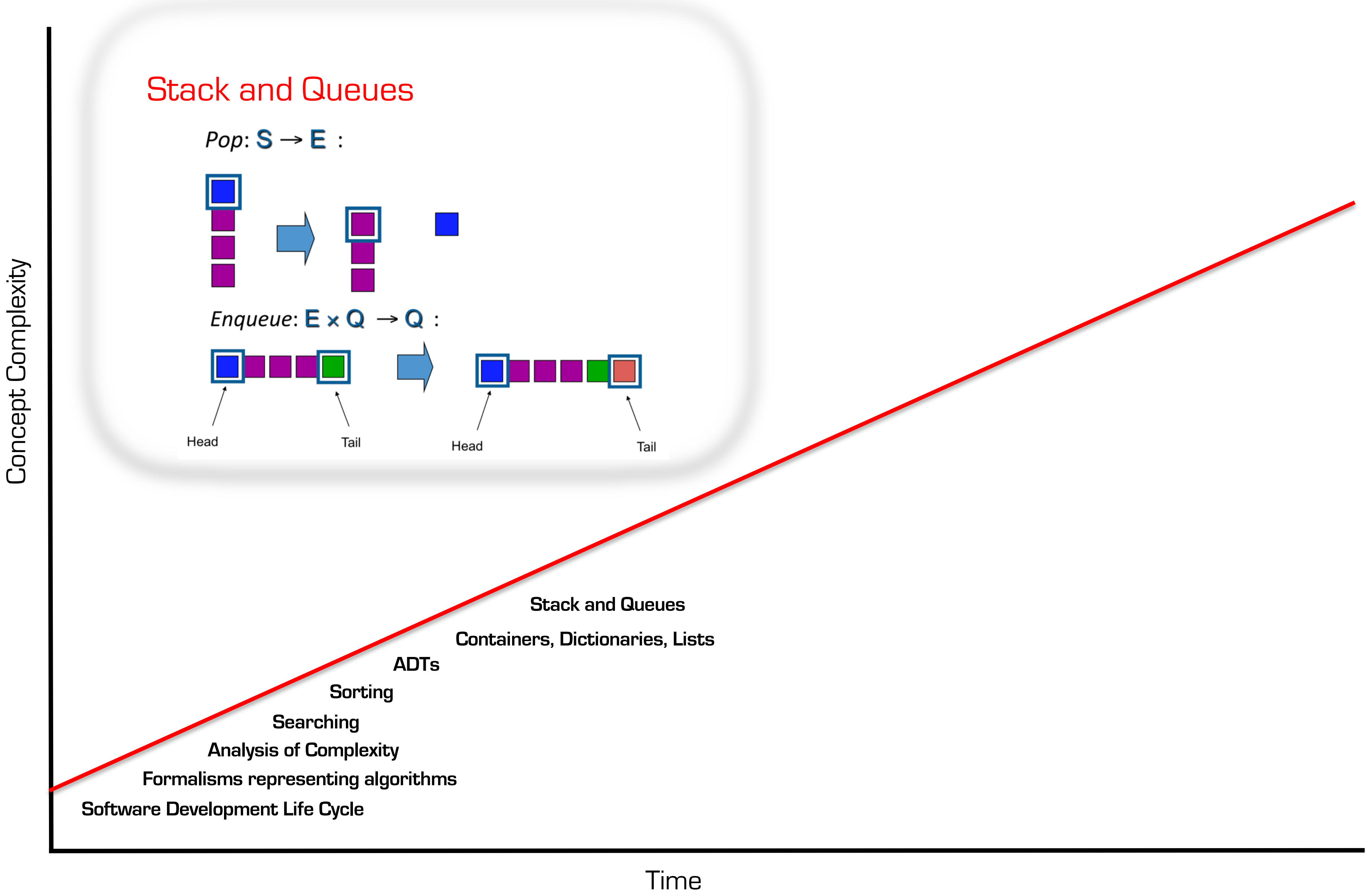
Data-Structures and Algorithms for Engineers

Containers, Dictionaries, & Lists

- ADT specification
- Array implementation
- Linked-list implementation



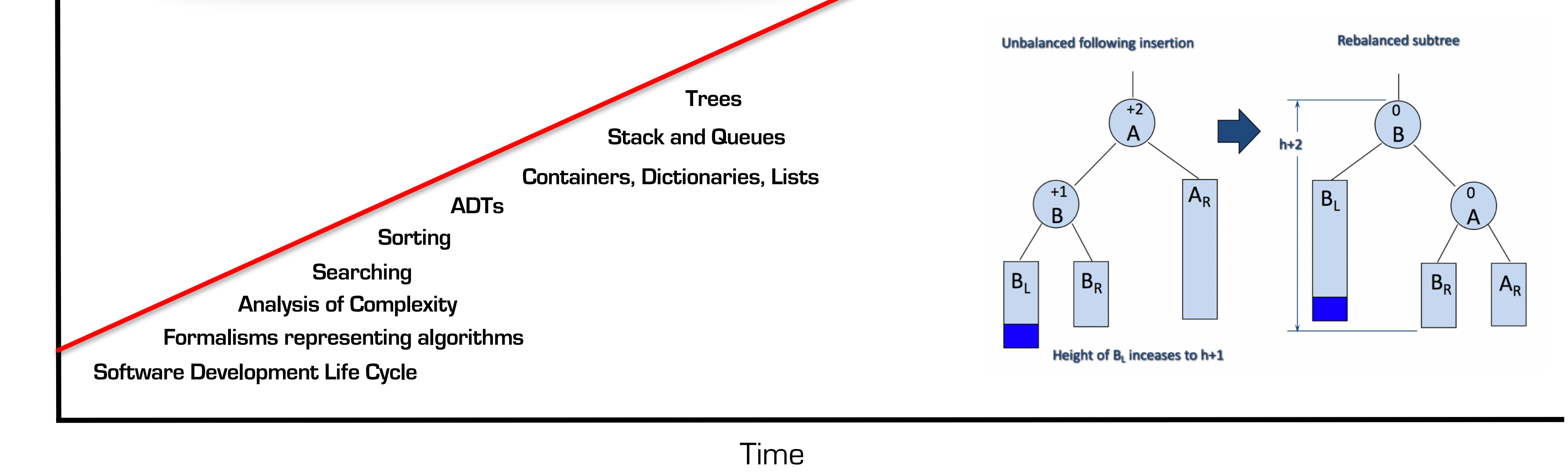
Data-Structures and Algorithms for Engineers



Data-Structures and Algorithms for Engineers

Trees

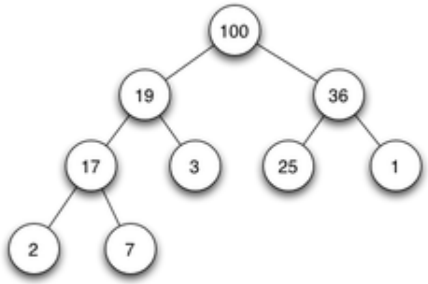
- Binary trees
- Binary search trees
- Tree traversal
- Applications of trees
  (e.g. Huffman coding)
- Height-balanced trees
  (e.g. AVL Trees, Red-Black Trees



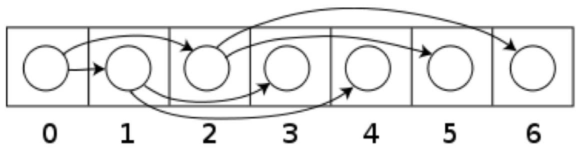
Data-Structures and Algorithms for Engineers

Heaps

Priority queues
Binary heaps, min/max-heaps
Heap operations
Heap sort



Heaps
Trees
Stack and Queues
Containers, Dictionaries, Lists
ADTs
Sorting
Searching
Analysis of Complexity
Formalisms representing algorithms
Software Development Life Cycle



Concept Complexity

Time

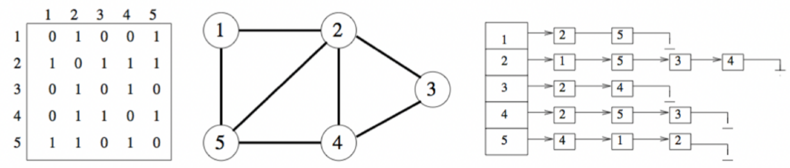
Data-Structures and Algorithms for Engineers

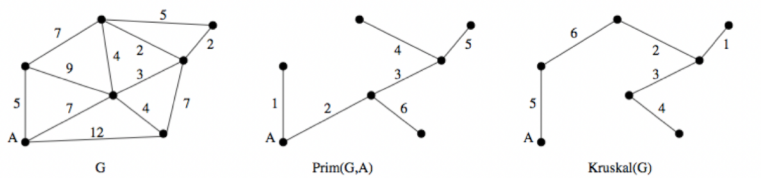
Graphs

- Types
- Representations
- BFS & DFS Traversals
- Topological sort
- Minimum spanning tree
  (e.g. Prim's and Kruskal's Algs.)
- Shortest-path algorithms
  (e.g. Dijkstra's & Floyd's Algs.)

Concept Complexity

Graphs
Heaps
Trees
Stack and Queues
Containers, Dictionaries, Lists
ADTs
Sorting
Searching
Analysis of Complexity
Formalisms representing algorithms
Software Development Life Cycle





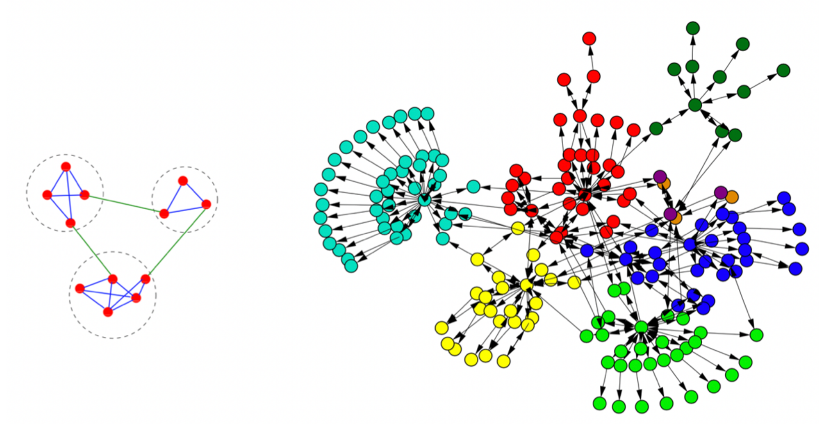
Time

Data-Structures and Algorithms for Engineers

Complex Networks

- Random networks
- Degree distribution
- Clustering
- Small world phenomena
- Scale free networks
- Community detection

Complex Networks
Graphs
Heaps
Trees
Stack and Queues
Containers, Dictionaries, Lists
ADTs
Sorting
Searching
Analysis of Complexity
Formalisms representing algorithms
Software Development Life Cycle



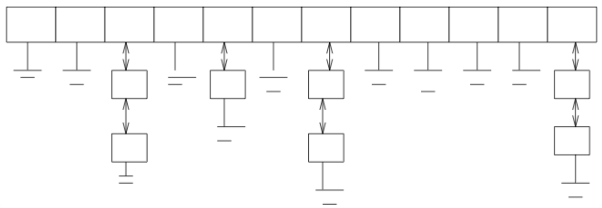
Concept Complexity

Time

Data-Structures and Algorithms for Engineers

Hashing

- Hash functions
- Collisions
- Chaining & Probe policies



Concept Complexity

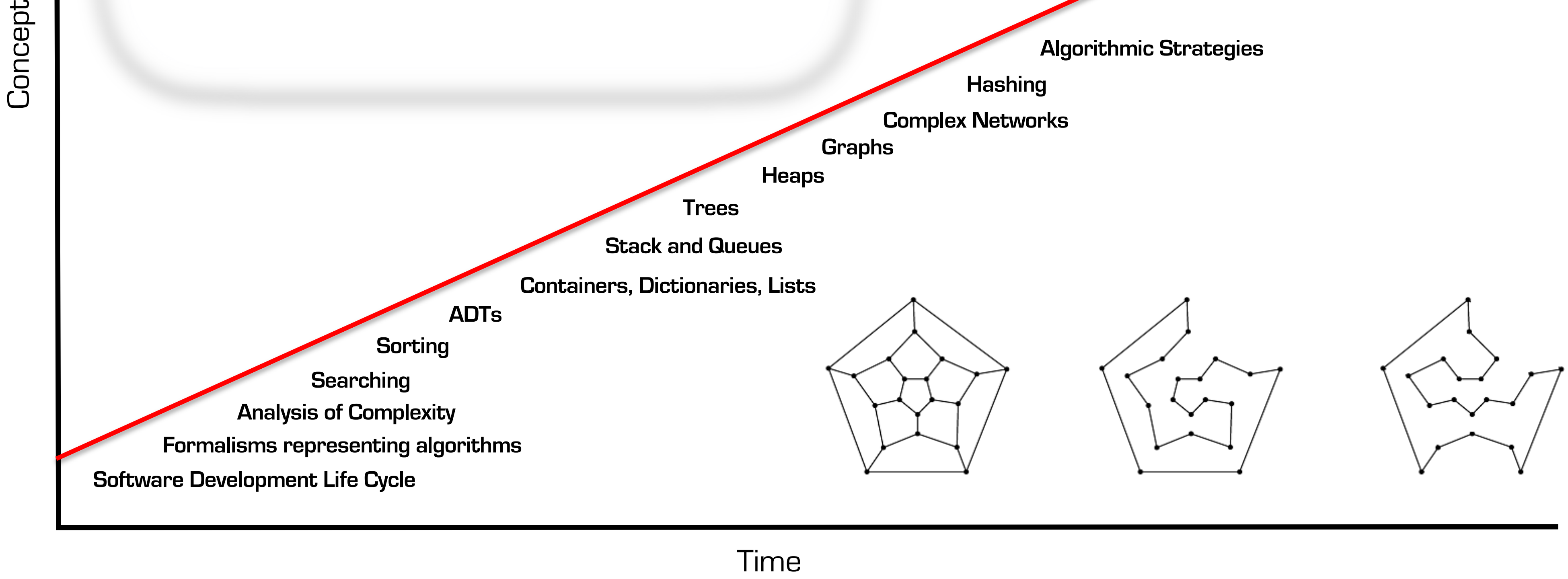
Software Development Life Cycle
Formalisms representing algorithms
Analysis of Complexity
Searching
Sorting
ADTs
Containers, Dictionaries, Lists
Stack and Queues
Trees
Heaps
Graphs
Complex Networks
Hashing

Time

Data-Structures and Algorithms for Engineers

Algorithmic Strategies

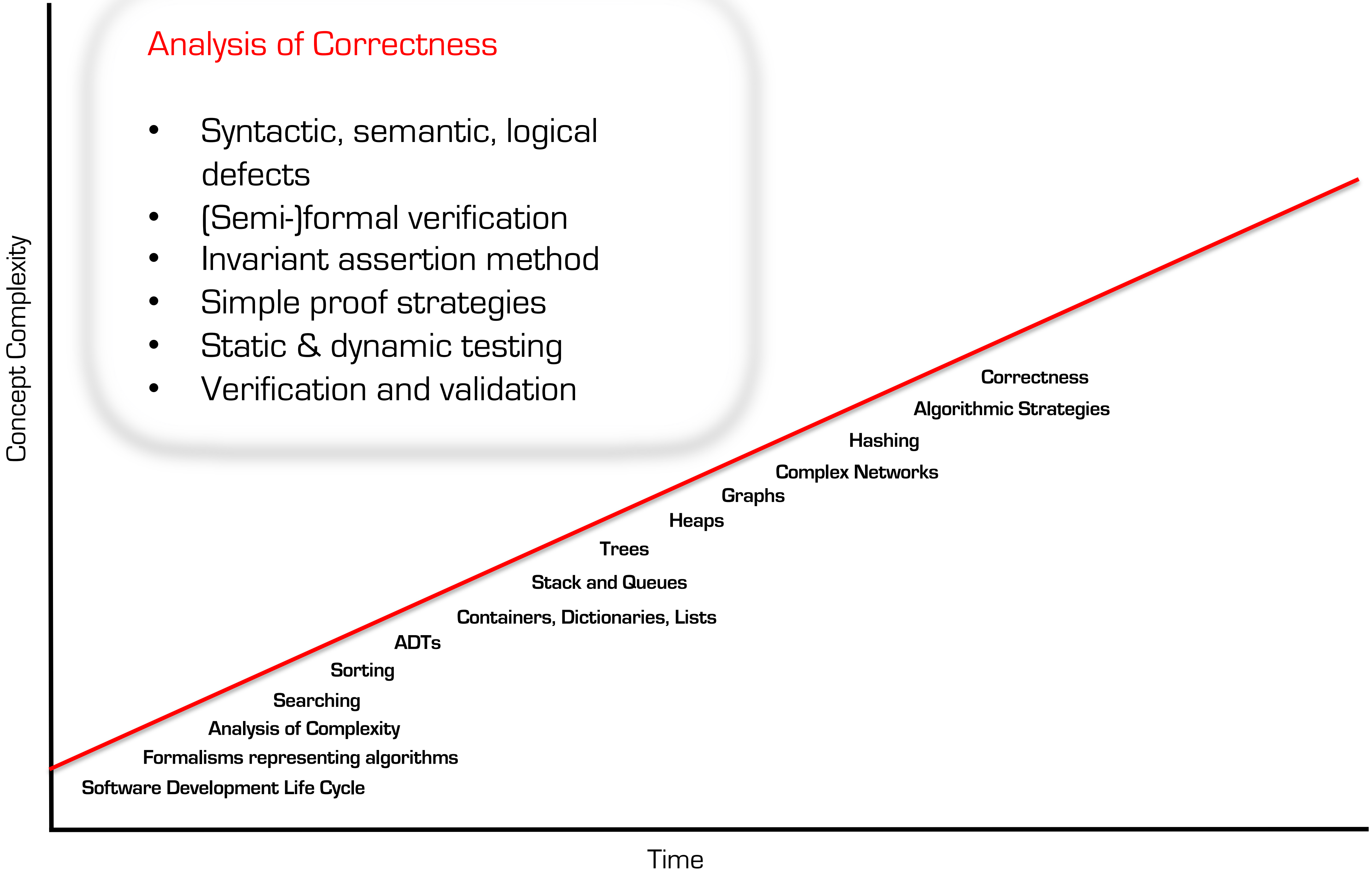
- Brute-force
- Divide-and-conquer
- Greedy algorithms
- Combinatorial Search
- Backtracking
- Branch-and-bound



Data-Structures and Algorithms for Engineers

Analysis of Correctness

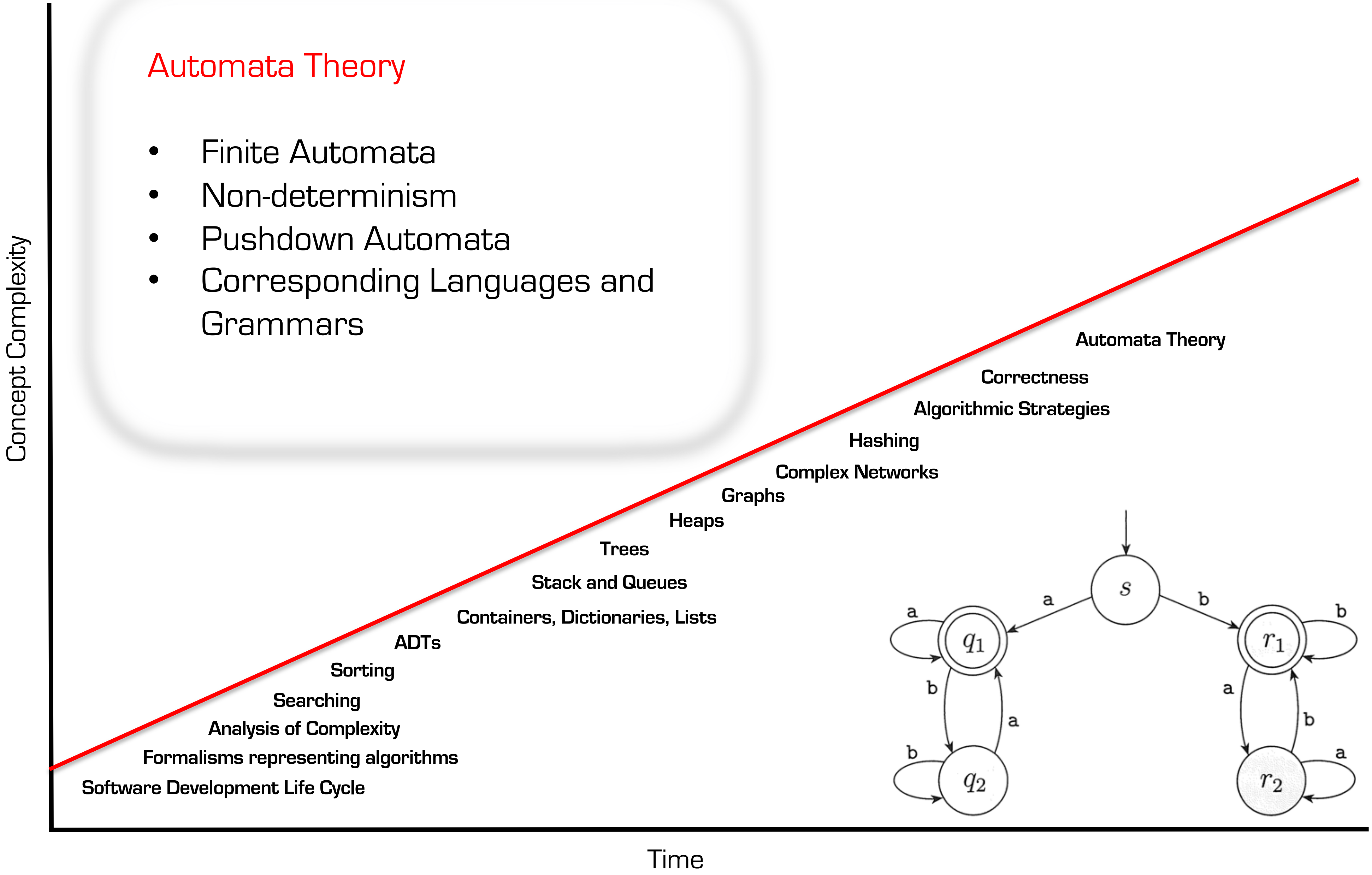
- Syntactic, semantic, logical defects
- (Semi-)formal verification
- Invariant assertion method
- Simple proof strategies
- Static & dynamic testing
- Verification and validation



Data-Structures and Algorithms for Engineers

Automata Theory

- Finite Automata
- Non-determinism
- Pushdown Automata
- Corresponding Languages and Grammars



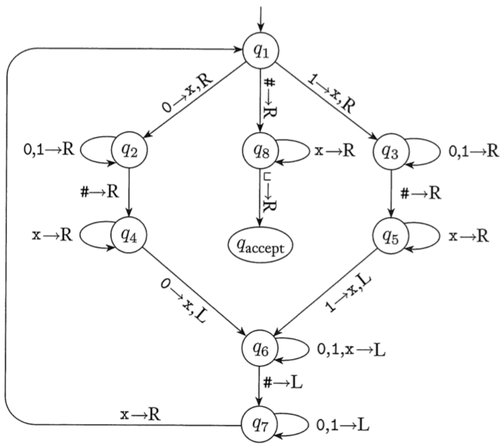
Data-Structures and Algorithms for Engineers

Computability Theory

- The Church-Turing Thesis
- Turing Machines
- The Definition of Algorithm
- Decidability
- Undecidability
- Reducibility

Concept Complexity

Computability Theory
Automata Theory
Correctness
Algorithmic Strategies
Hashing
Complex Networks
Graphs
Heaps
Trees
Stack and Queues
Containers, Dictionaries, Lists
ADTs
Sorting
Searching
Analysis of Complexity
Formalisms representing algorithms
Software Development Life Cycle



Time

Software Development Tools for Exercises and Assignments

Software Development Tools for Exercises and Assignments

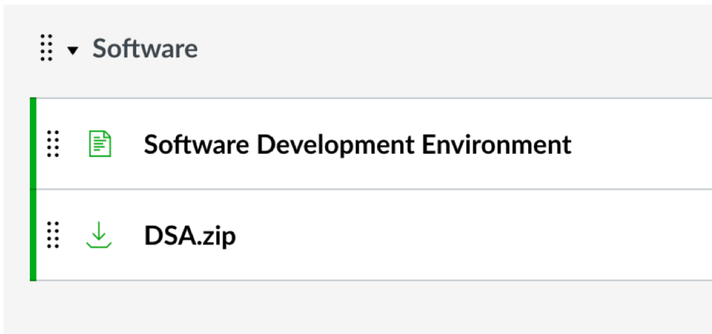
- Installation of software development environment
  - Windows 10 OS
  - Microsoft Visual C++ Express compiler, version 10.0 (also known as Visual C++ 2010 or MSVC++ 2010)
  - Cmake
  - DSA Repository

- Let's walk through the process for installing these tools …

https://canvas.cmu.edu/courses/3210/modules

Software

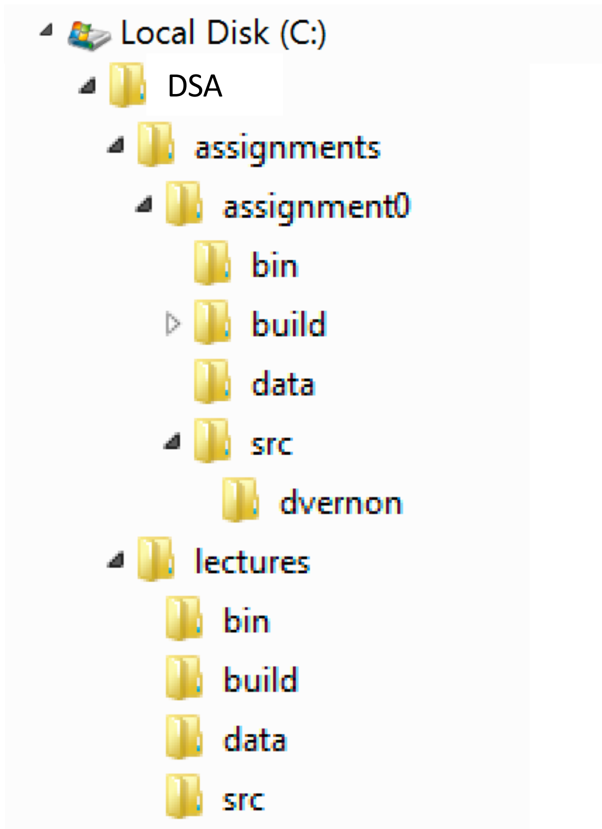
Software Development Environment



Software Development Tools for Exercises and Assignments

- Installation of software development environment
  - `C:\DSA`
  - Fixed file organization
- Let's walk through the process to compile and run the program in

  `C:\DSA\assignments\assignment0\dvernon`



Software Development Tools for Exercises and Assignments

- Preferred practice for software that supports encapsulation and data hiding (e.g. ADT & OO classes)

- 3 files: Interface, Implementation, and Application Files
  - Interface
    - between implementation and application
    - Header File that declares the class type
    - Functions, classes, are declared, not defined (except inline functions)

  - Implementation
    - `#includes` the interface file
    - contains the function definitions

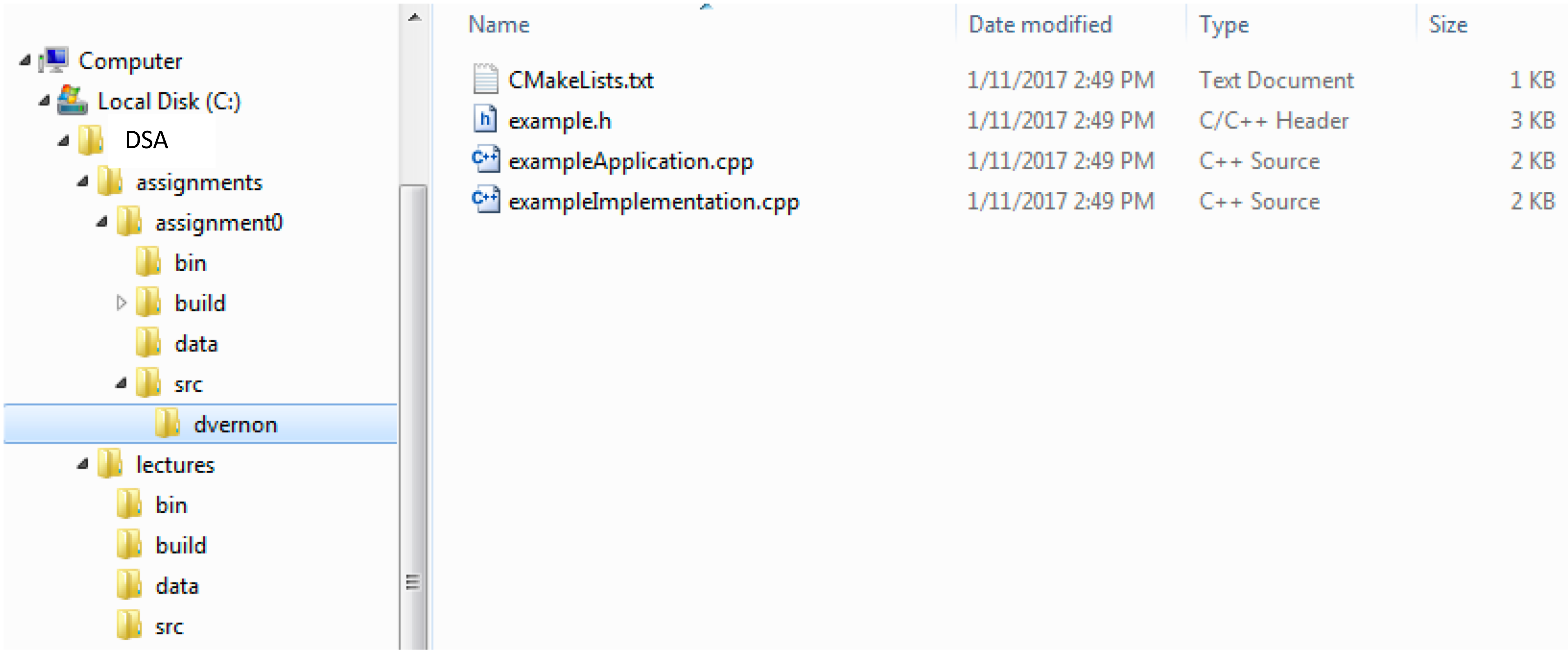
  - Application
    - `#includes` the interface file
    - contains other (application) functions, including the `main` function

Software Development Tools for Exercises and Assignments

When writing an application, we are ADT/class users

- Should not know about the implementation of the ADT/class
- Thus, the interface must furnish all the necessary information to use the ADT/class
  - It also needs to be very well documented (internally)
- Also, the implementation should be quite general (cf. reusability)

Software Development Tools for Exercises and Assignments



Software Development Tools for Exercises and Assignments

Exercises

- Install software development tools
- Install DSA repository
- Compile and run the program in

```
C:\DSA\assignments\assignment0\dvernon
```

- Create, compile and run a new program in

```
C:\DSA\assignments\assignment0\myandrewid
```

Replace with your Andrew Id

Software Development Tools for Exercises and Assignments

- For your first assignment, you will simply copy the **`assignment0`** directory to **`assignment1`** and follow a similar compilation procedure, writing new assignment-specific code.

- There is just one thing you need to do: edit the

  `C:DSA\assignments\assignment1\CMakeLists.txt`

  and change the project name from assignment0 to assignment1, viz:

```
###############################################
PROJECT(assignment0)
###############################################
```

Becomes

```
###############################################
PROJECT(assignment1)
###############################################
```

Software Development Tools for Exercises and Assignments

When submitting an assignment, all you have to do is submit a zip version of your `myandrewid` directory containing

- Your three source code files
- The CmakeLists.txt file
- The input.txt file (copied from the data directory)
- The output.txt file (copied from the data directory)

Levels of Abstraction in Information Processing Systems

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

The New York Times

PROFILES IN SCIENCE

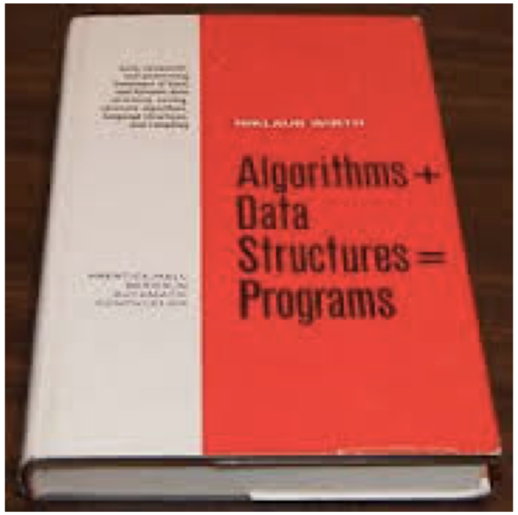
The Yoda of Silicon Valley

Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, "The Art of Computer Programming."

Listed by American Scientist in 2013 as one of the books that shaped the last century of science

https://www.nytimes.com/2018/12/17/science/donald-knuth-computers-algorithms-programming.html

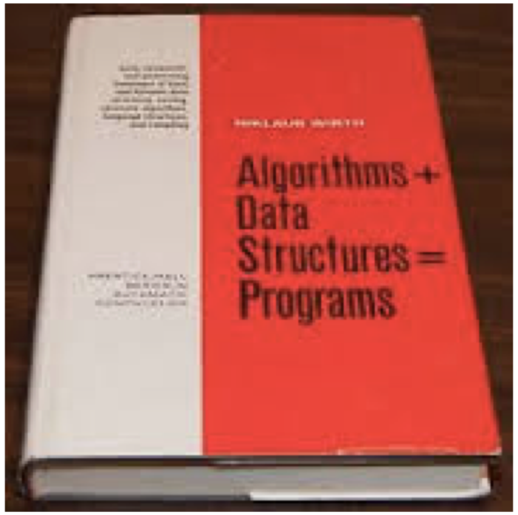
Algorithms + Data Structures = Programs



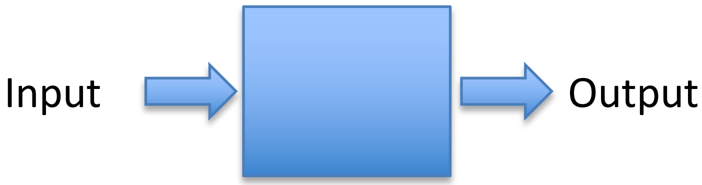
Niklaus Wirth, 1976

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

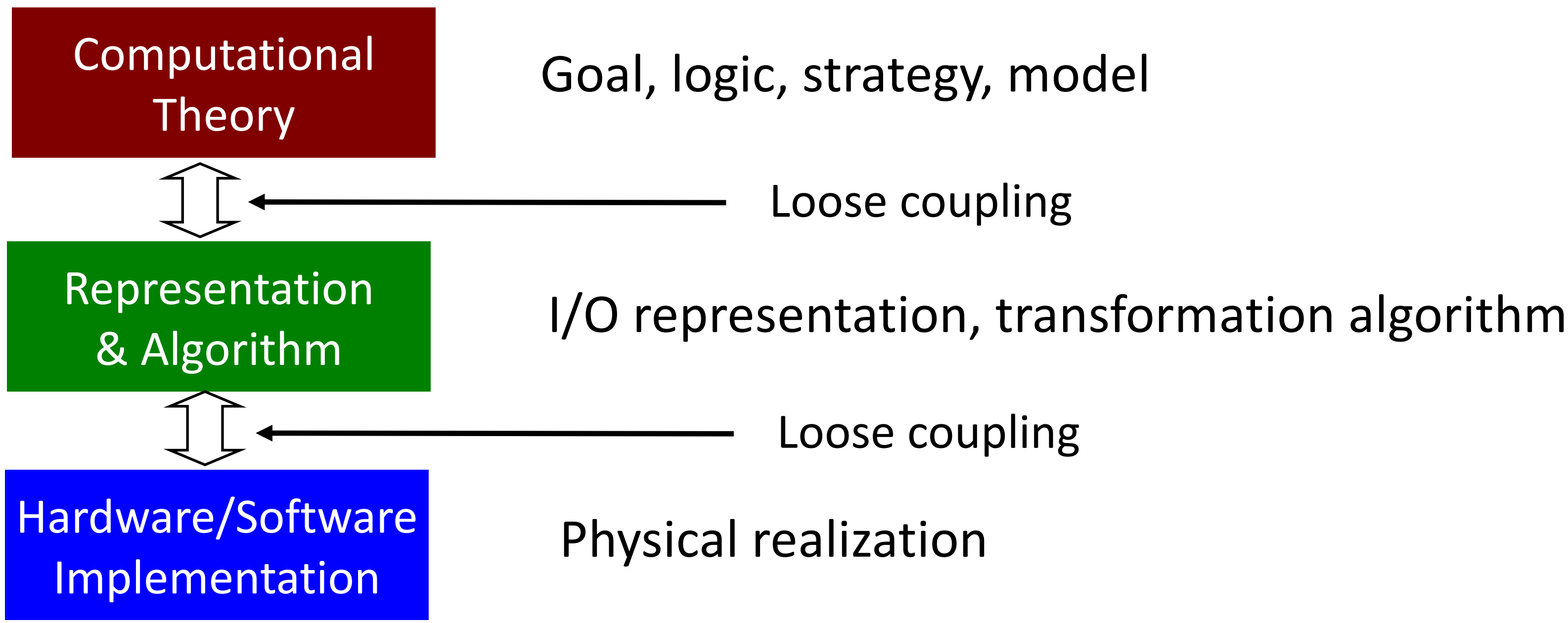
1969



Information Processing:
Representation & Transformation



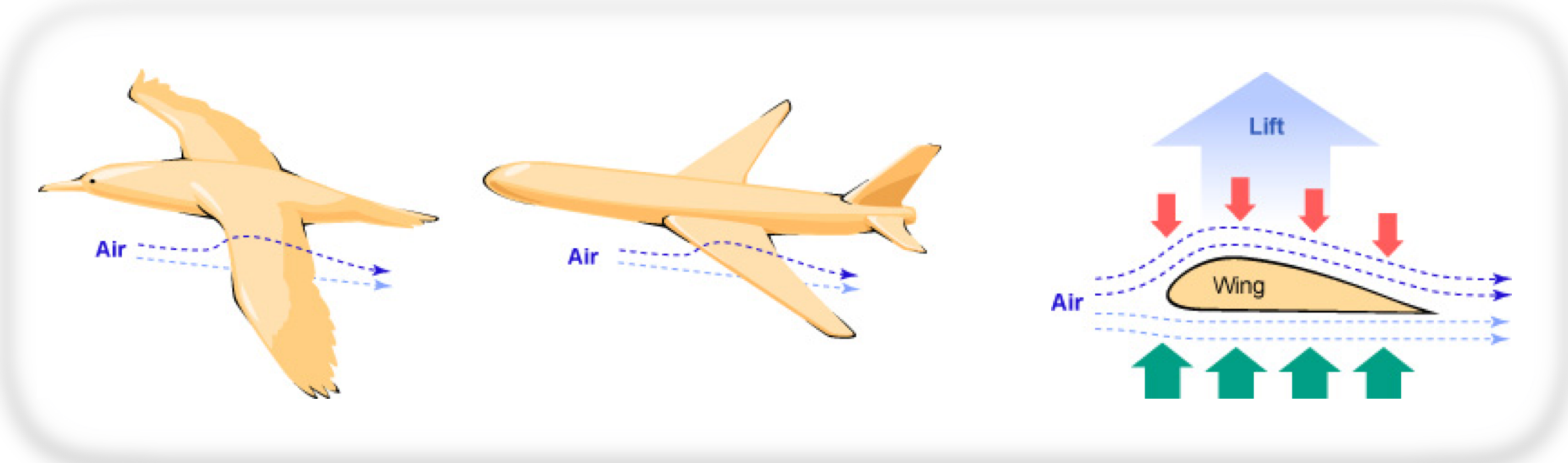
Marr’s Hierarchy of Abstraction / Levels of Understanding Framework

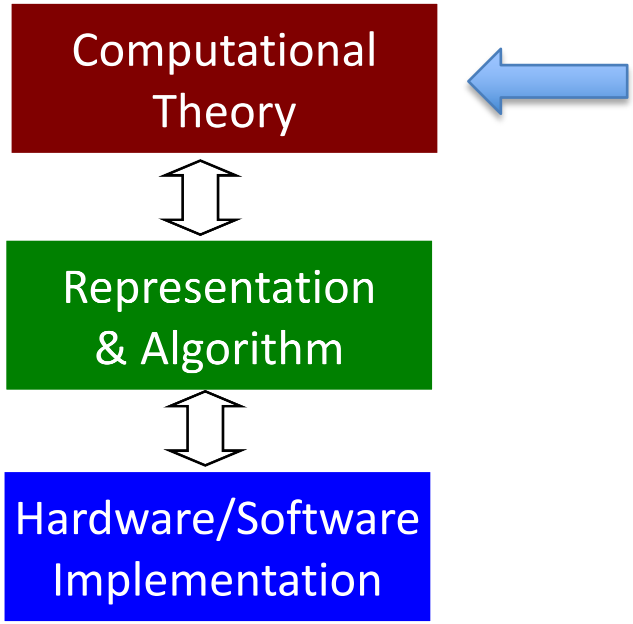


Marr’s Hierarchy of Abstraction / Levels of Understanding Framework

“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.



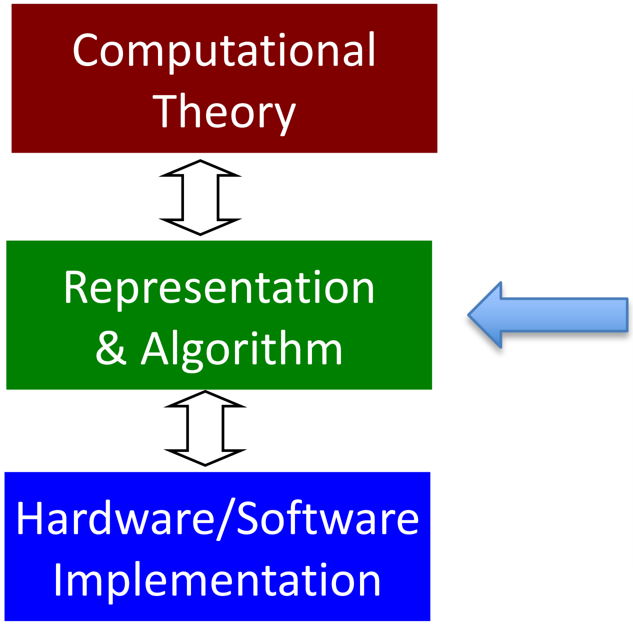


Sorting a List

Given a sequence of n keys $a_1, \ldots , a_n$

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

$1 \leq i, j \leq n$



Sorting a List

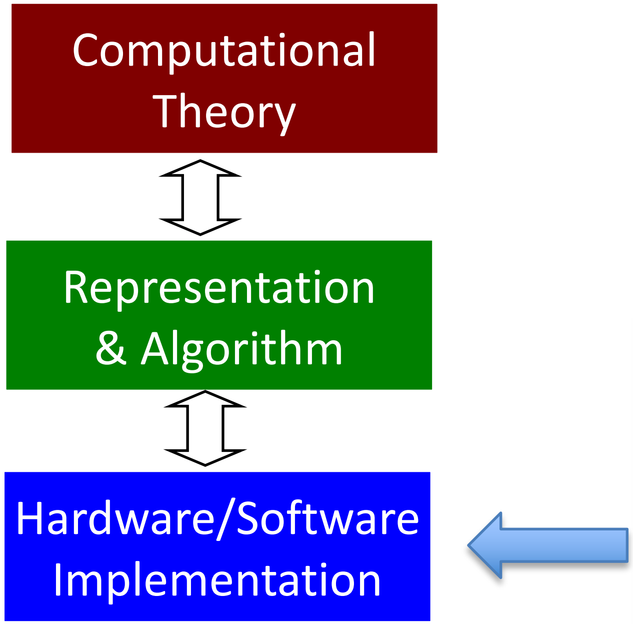
Bubble Sort

Insertion Sort

Quick Sort

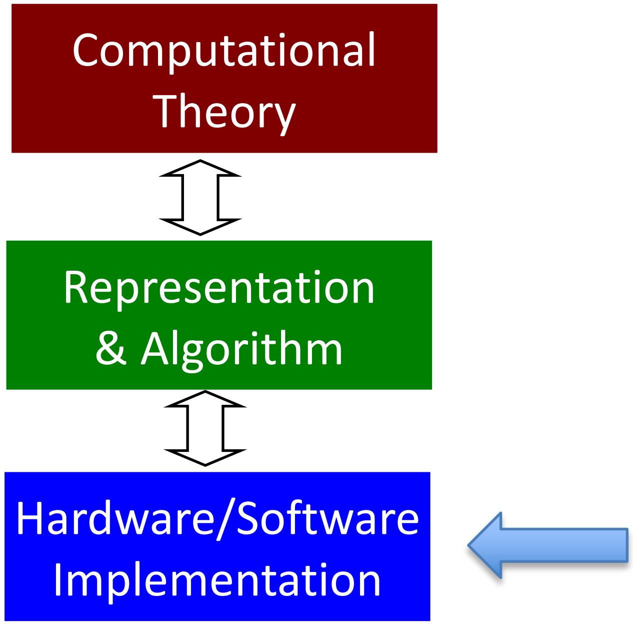
Merge Sort, …

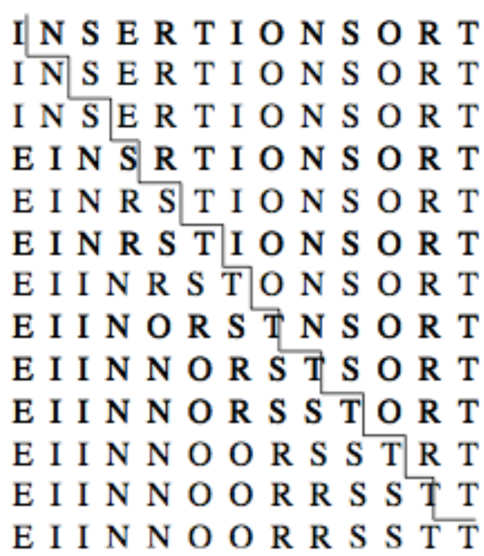
Key point: different computational efficiency

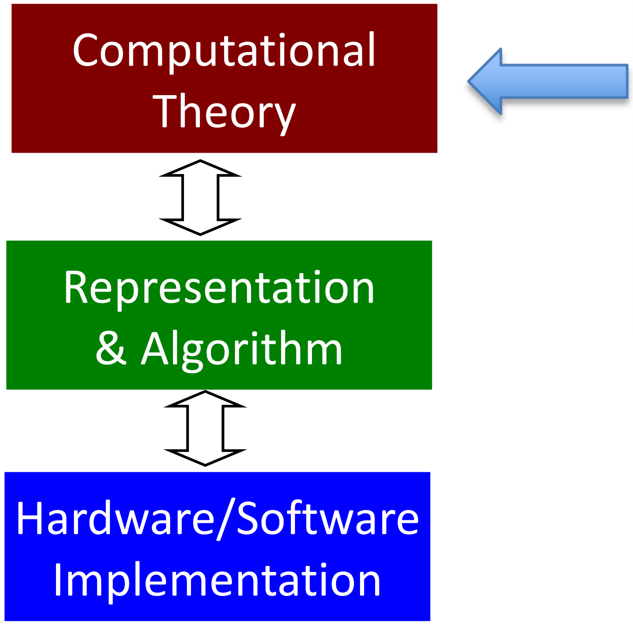


Sorting a List

```
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

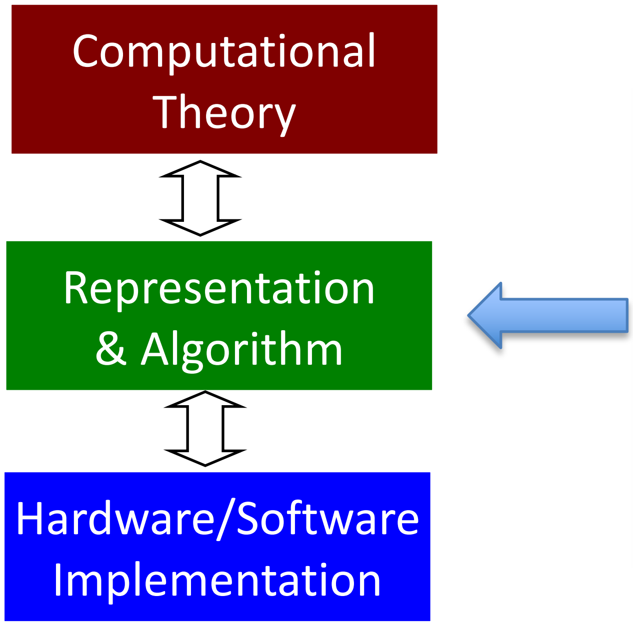

Sorting a List
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T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

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

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



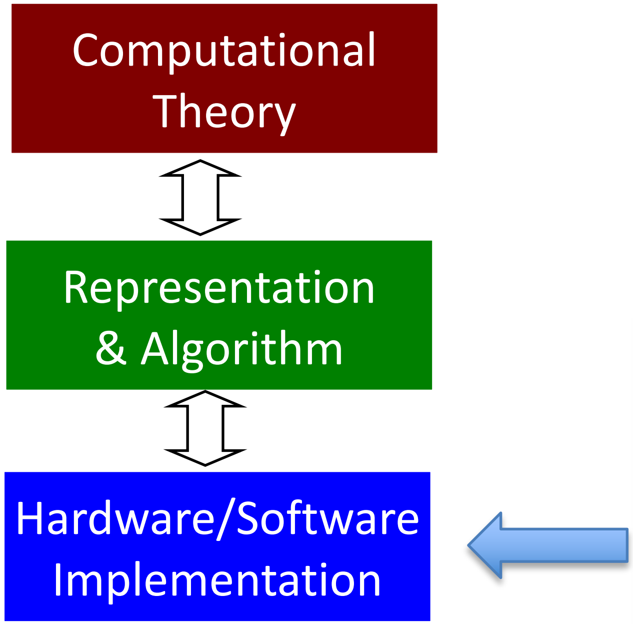
Fourier Transform

DFT: Discrete Fourier Transform

FFT: Fast Fourier Transform

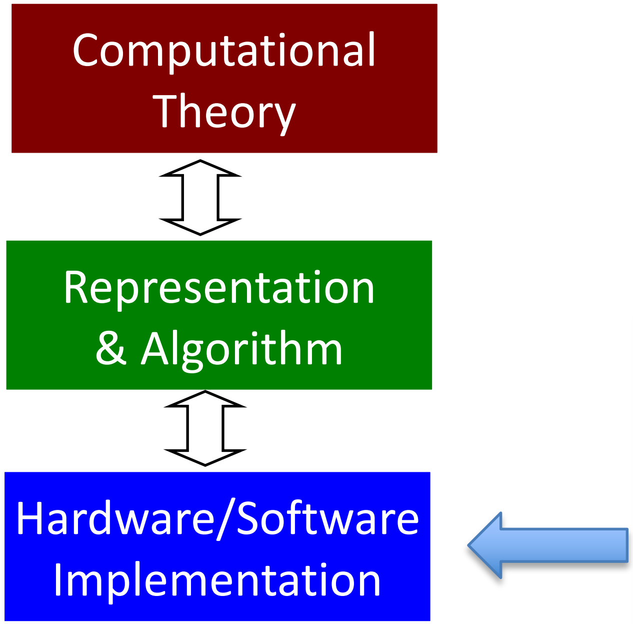
FFTW: Fasted Fourier Transform in the West

Key point: different computational efficiency

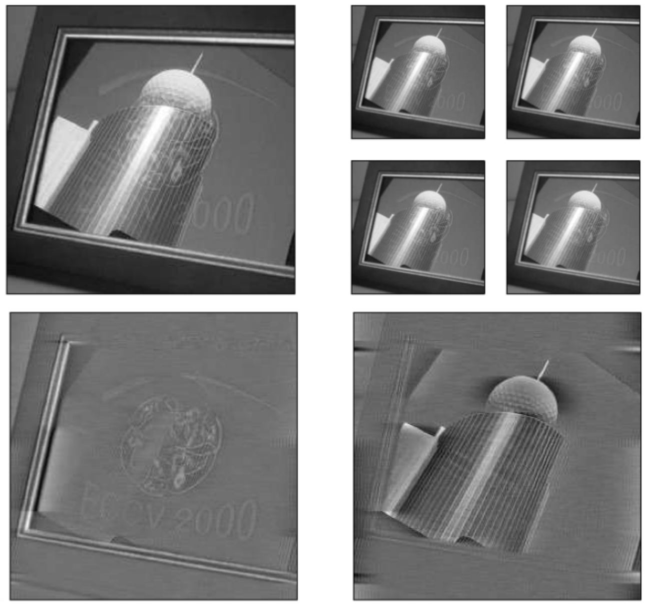


Fourier Transform

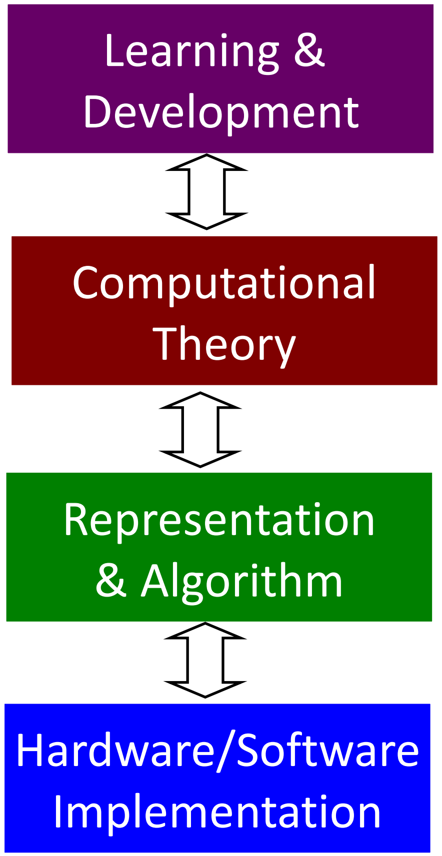
```
main()
{
        unsigned long i;
        int isign;
        float *data1,*data2,*fft1,*fft2;

        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

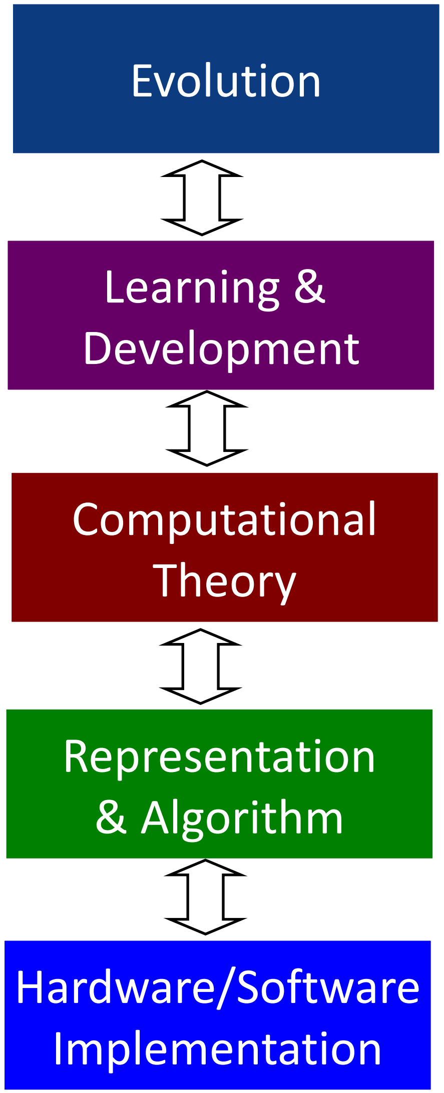


Marr's Levels of Understanding Framework updated 2012 by T. Poggio



Calibrating & improving the model

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

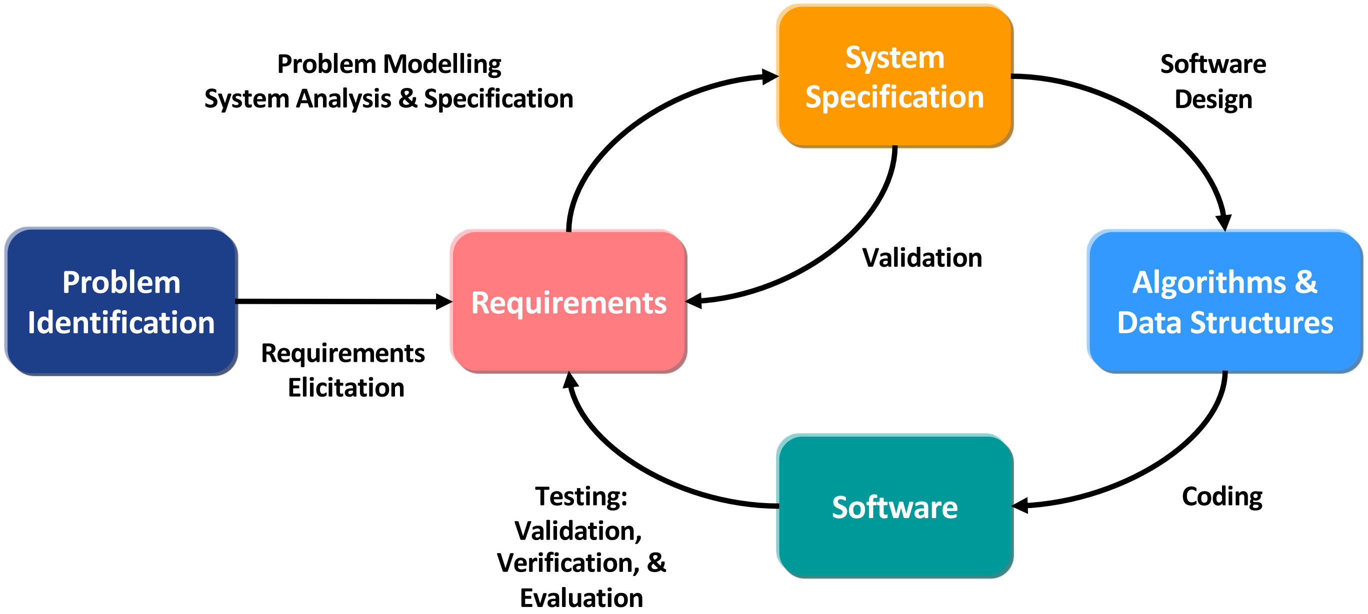


Generating new models

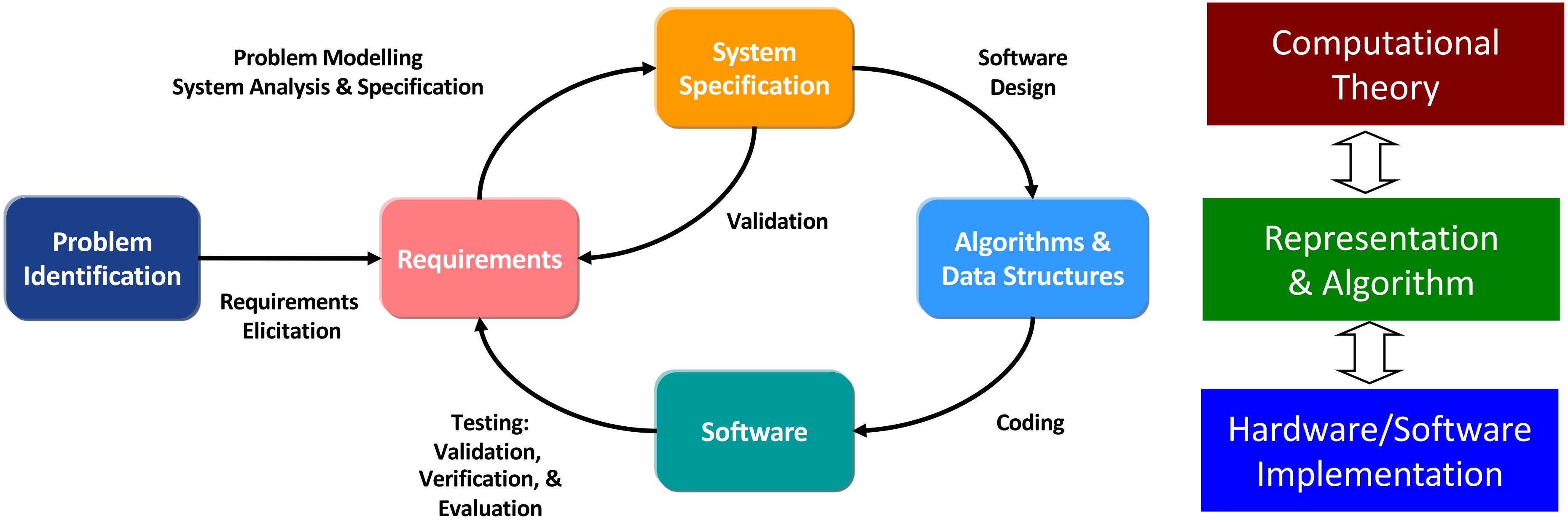
Calibrating & improving the model

The Software Development Life Cycle

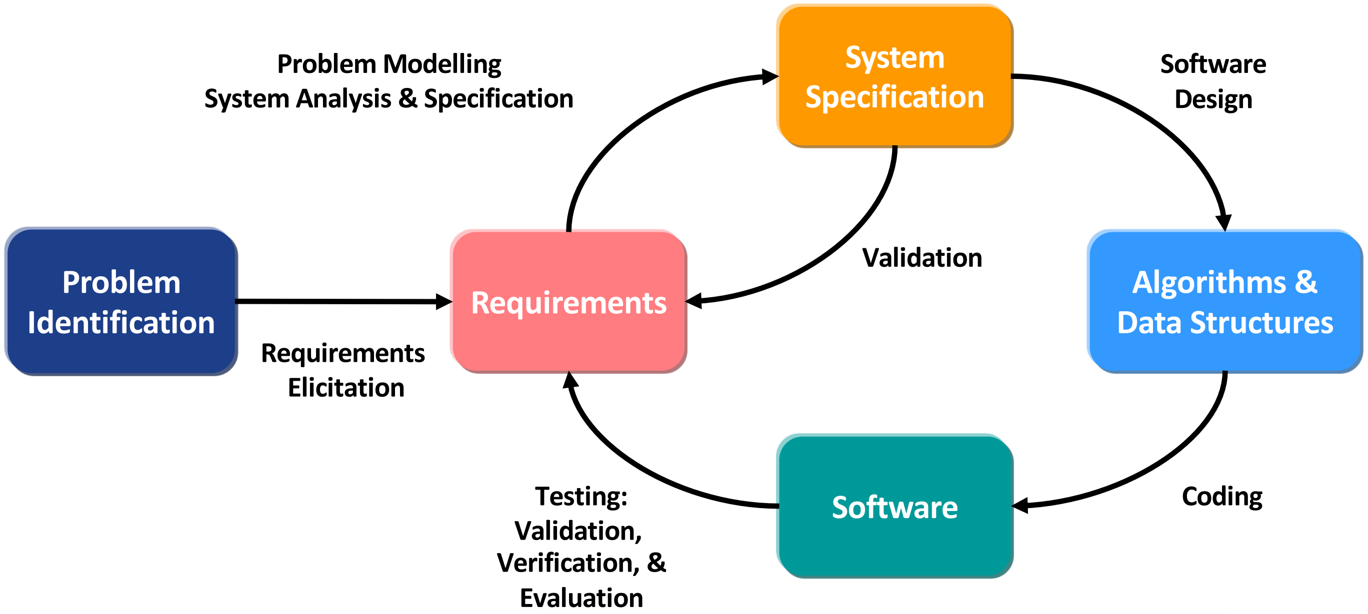
The Software Development Life Cycle

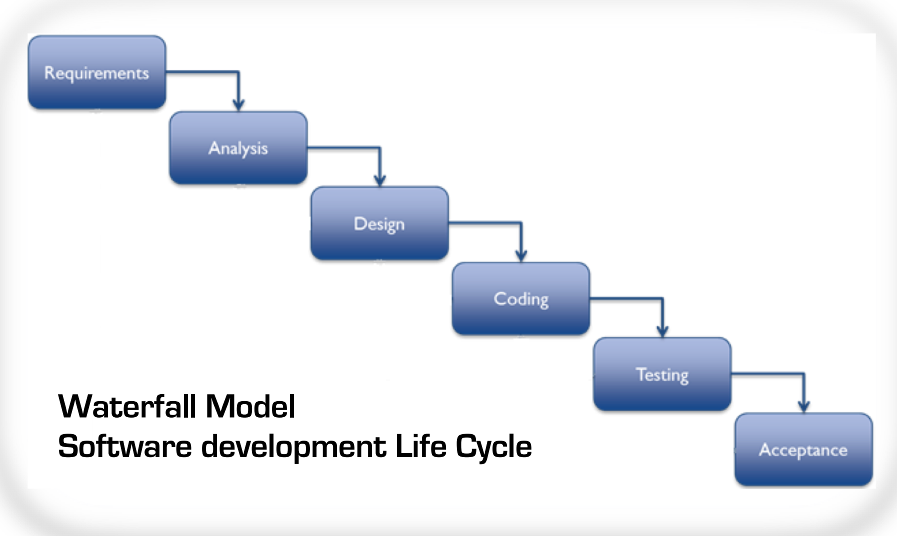


The Software Development Life Cycle

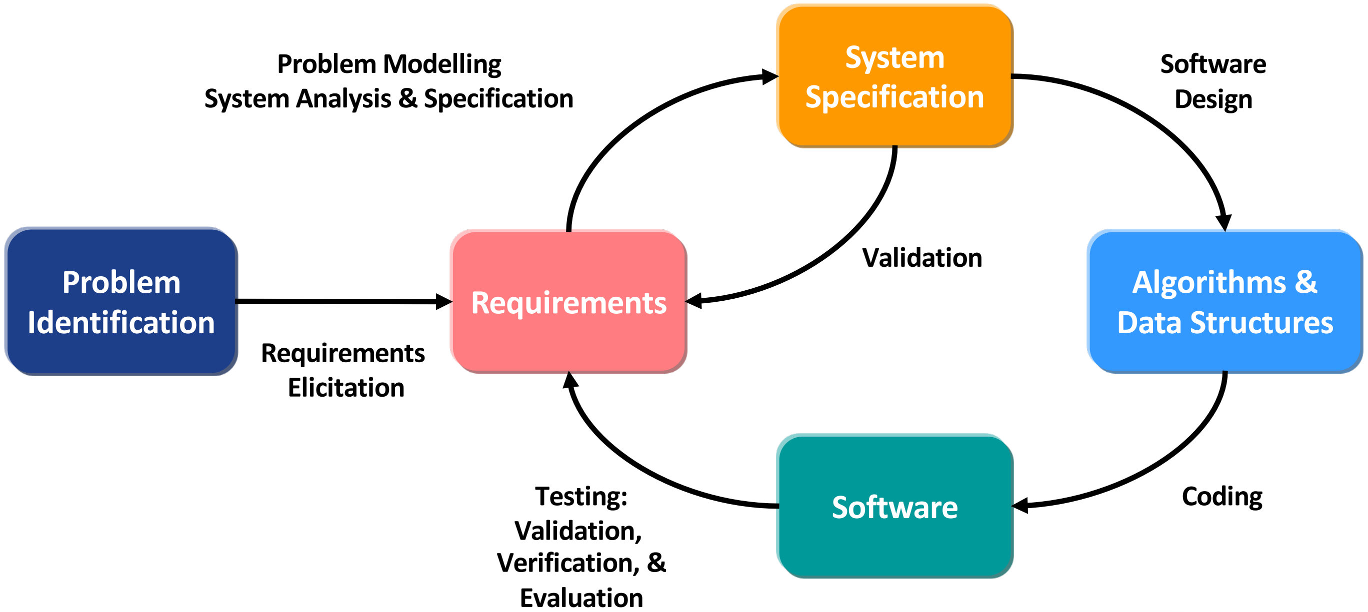


The Software Development Life Cycle





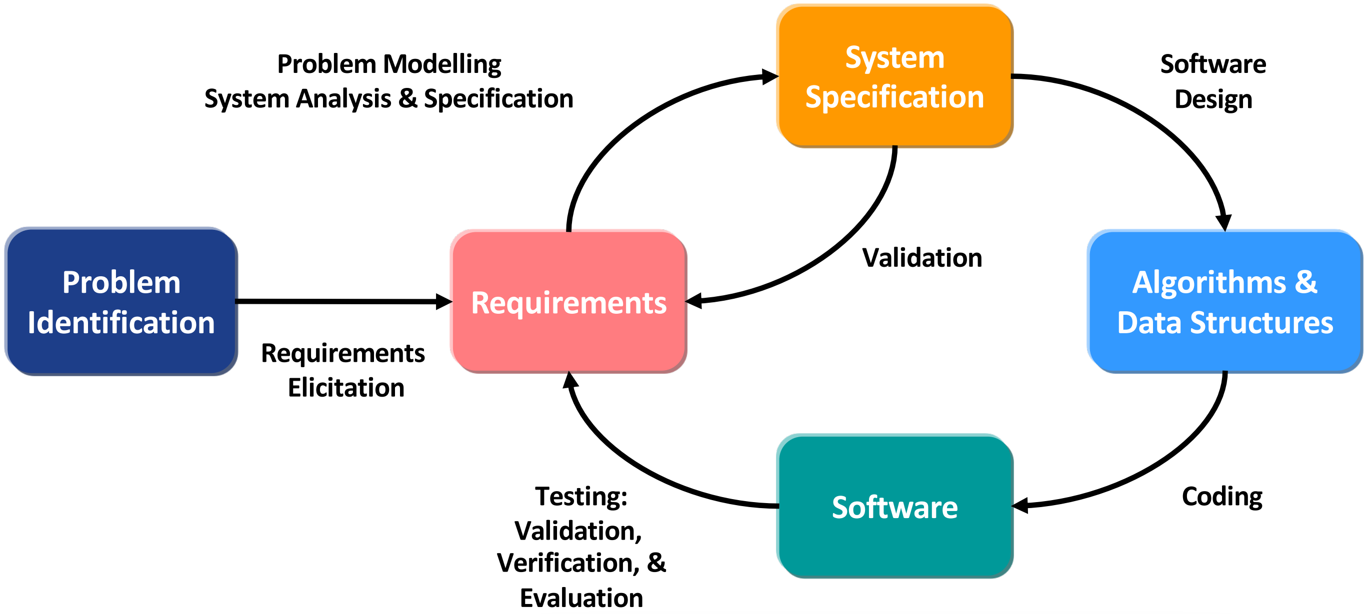
The Software Development Life Cycle



Life Cycle Models (Software Process Models):

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

The Software Development Life Cycle



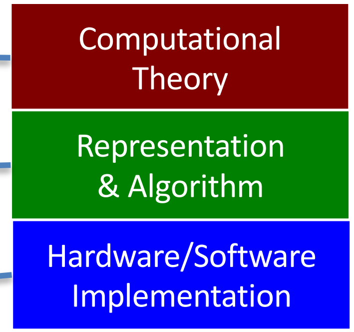
Software Development Methodologies:

Top-down
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

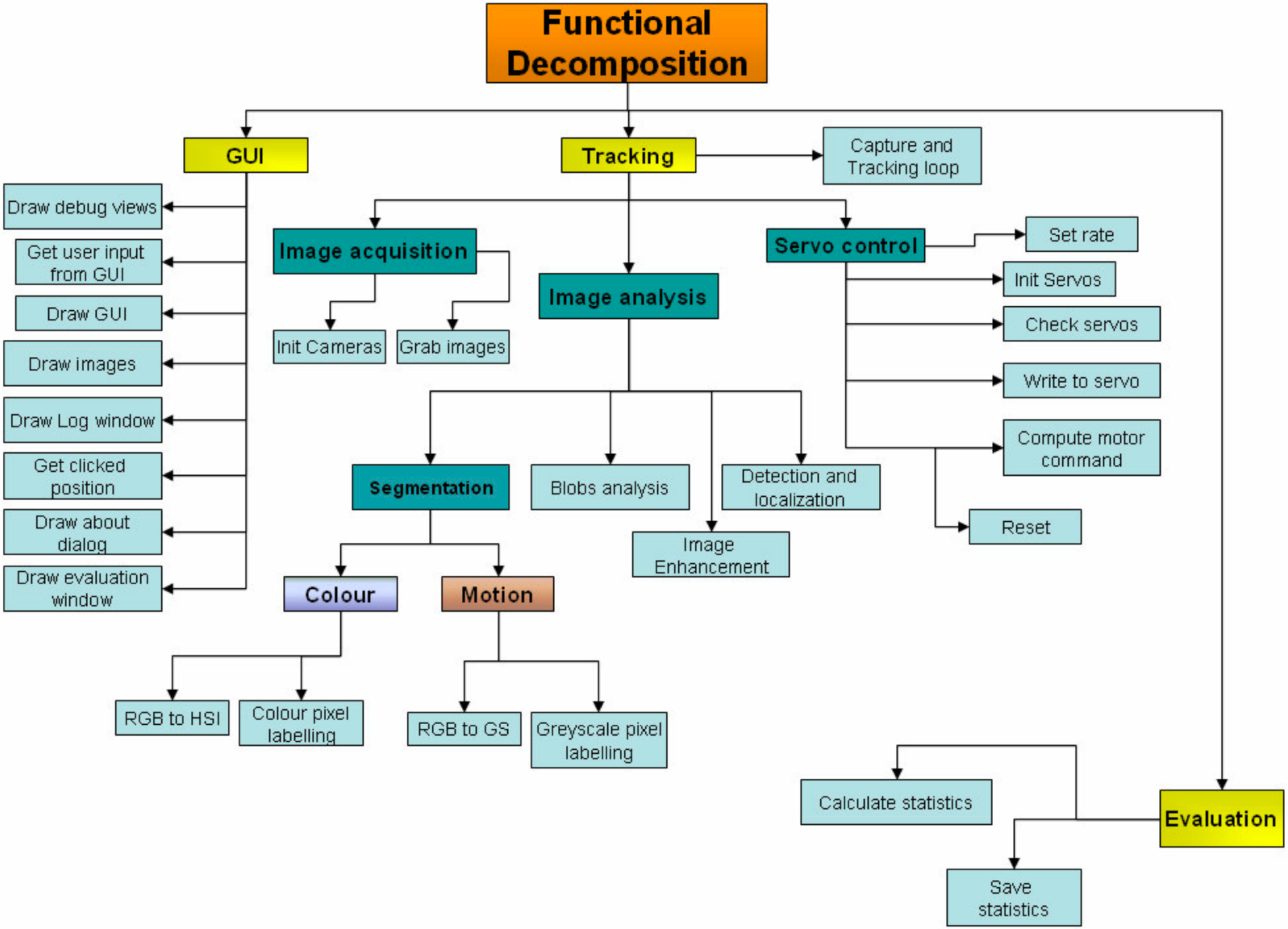
   – Including
     - Functional model
     - Data model
     - Process-flow model
     - Behavioural model

Software Development Life Cycle

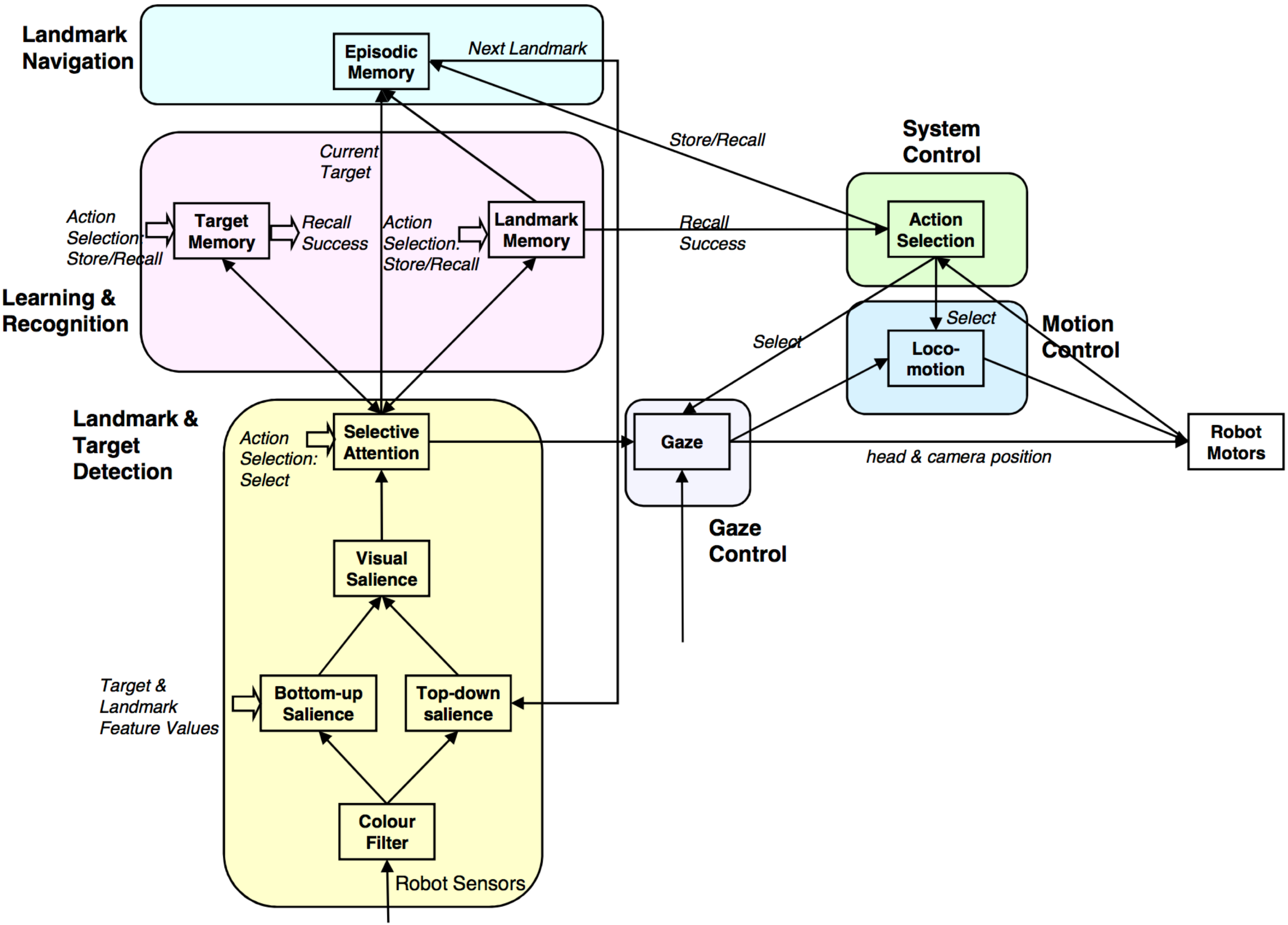
4. System analysis & specification

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



Functional Decomposition
GUI
Tracking
Capture and Tracking loop
Draw debug views
Get user input from GUI
Draw GUI
Draw images
Draw Log window
Get clicked position
Draw about dialog
Draw evaluation window
Image acquisition
Init Cameras
Grab images
Image analysis
Servo control
Set rate
Init Servos
Check servos
Write to servo
Compute motor command
Reset
Segmentation
Blobs analysis
Detection and localization
Image Enhancement
Colour
Motion
RGB to HSI
Colour pixel labelling
RGB to GS
Greyscale pixel labelling
Calculate statistics
Evaluation
Save statistics



Landmark Navigation
Episodic Memory
Next Landmark
Store/Recall
System Control
Current Target
Action Selection Store/Recall
Target Memory
Recall Success
Action Selection: Store/Recall
Landmark Memory
Recall Success
Action Selection
Learning & Recognition
Select
Select
Loco-motion
Motion Control
Landmark & Target Detection
Action Selection: Select
Selective Attention
Gaze
head & camera position
Robot Motors
Gaze Control
Visual Salience
Target & Landmark Feature Values
Bottom-up Salience
Top-down salience
Colour Filter
Robot Sensors

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

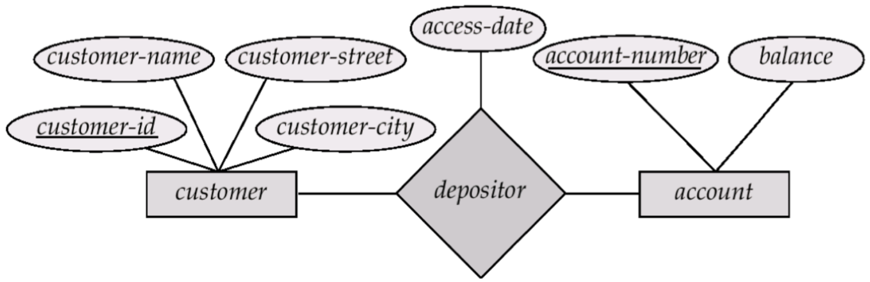
Also responsible for the concepts of data hiding and encapsulation, cf. ADT in Lecture 5

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 entites
- 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

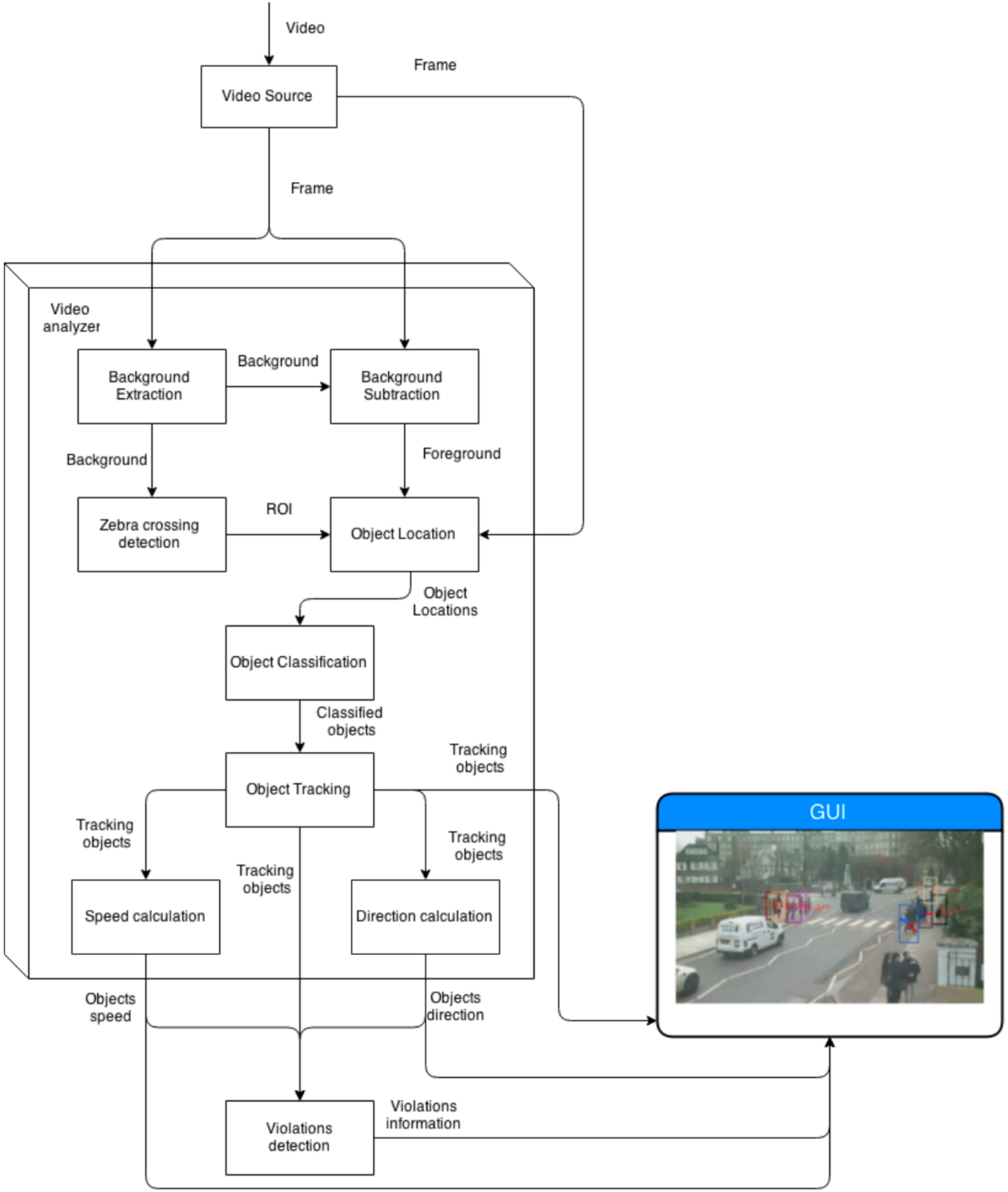
   – DFDs model the transformation of inputs into outputs

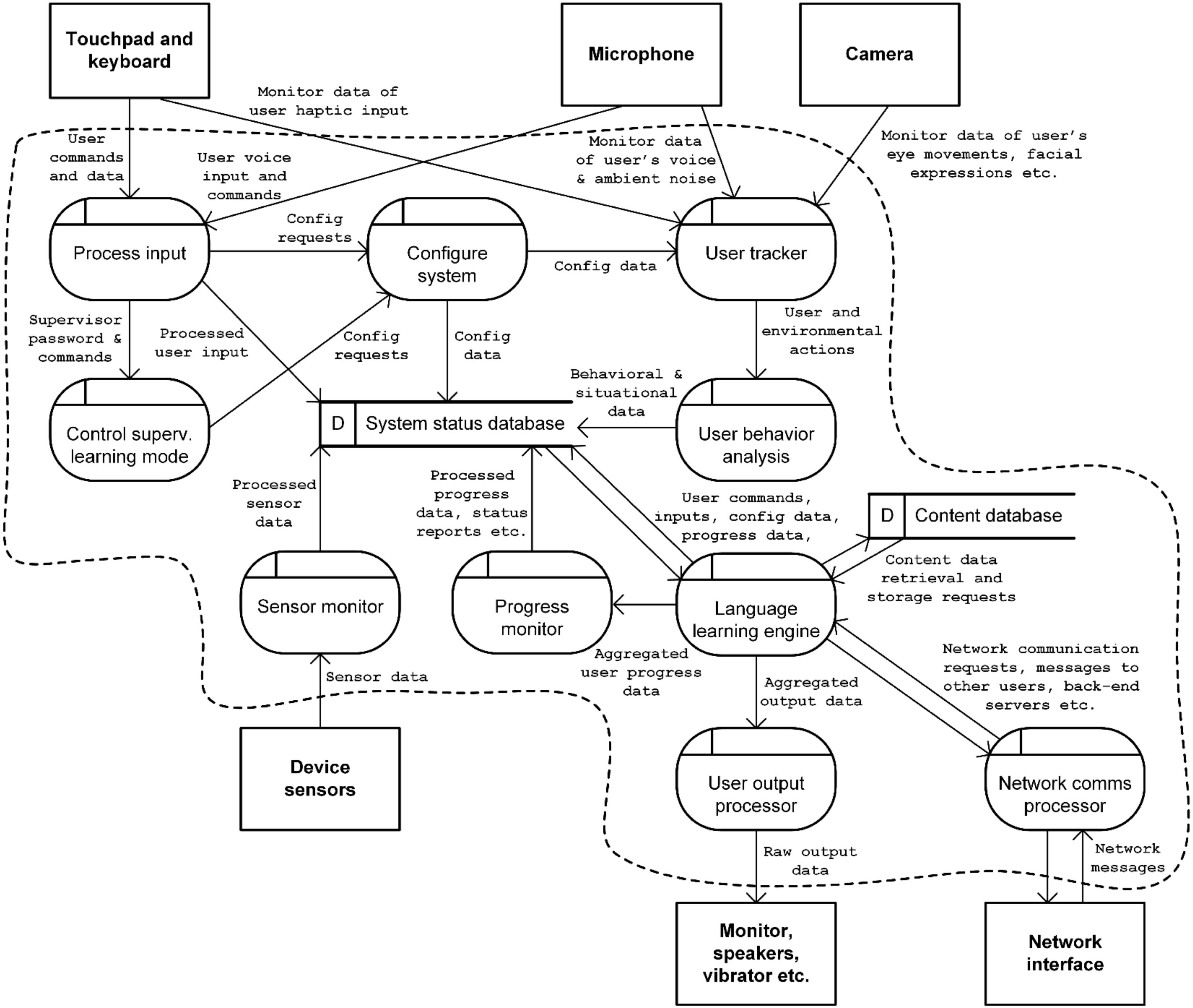
   – Processes/Functions represent individual functions that the system carries out and transform inputs to outputs

   – 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



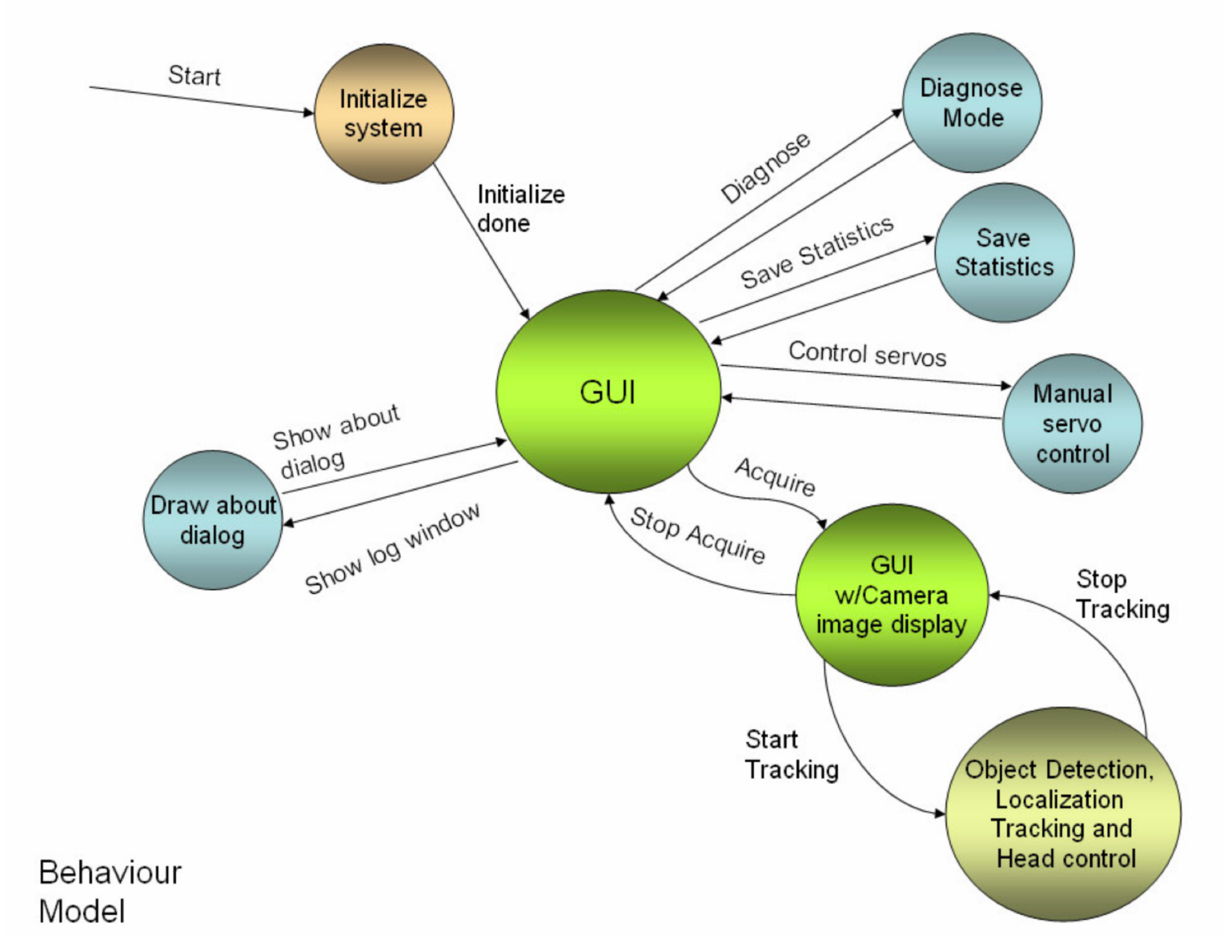
Touchpad and keyboard
Microphone
Camera
Monitor data of user haptic input
User commands and data
User voice input and commands
Monitor data of user's voice & ambient noise
Monitor data of user's eye movements, facial expressions etc.
Process input
Config requests
Configure system
Config data
User tracker
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

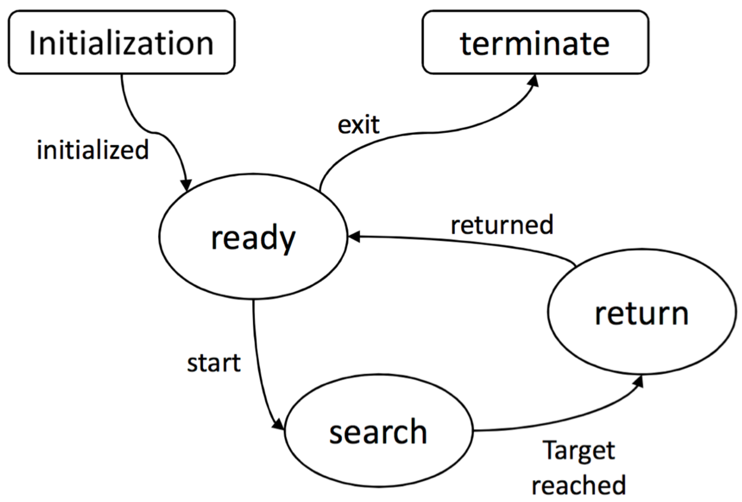
4. System analysis & specification

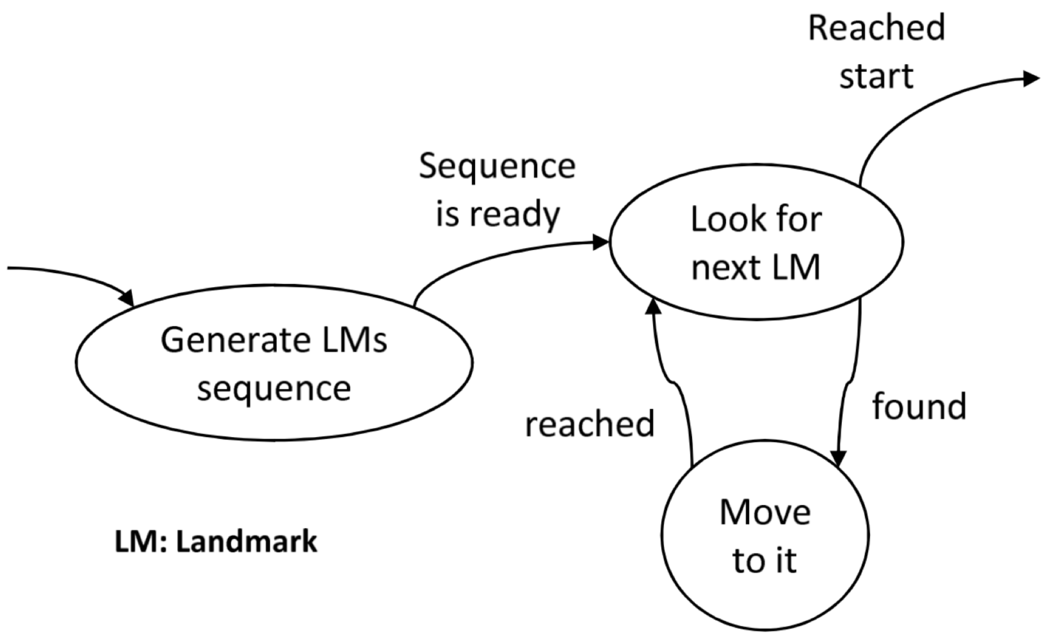
   Behavioural model

   - 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
- Data-structures
- Files
- Interface protocols

Effect the functional I/O transformation, i.e. realize computational theory

Representation of the input, temporary, and output data

– 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.

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 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

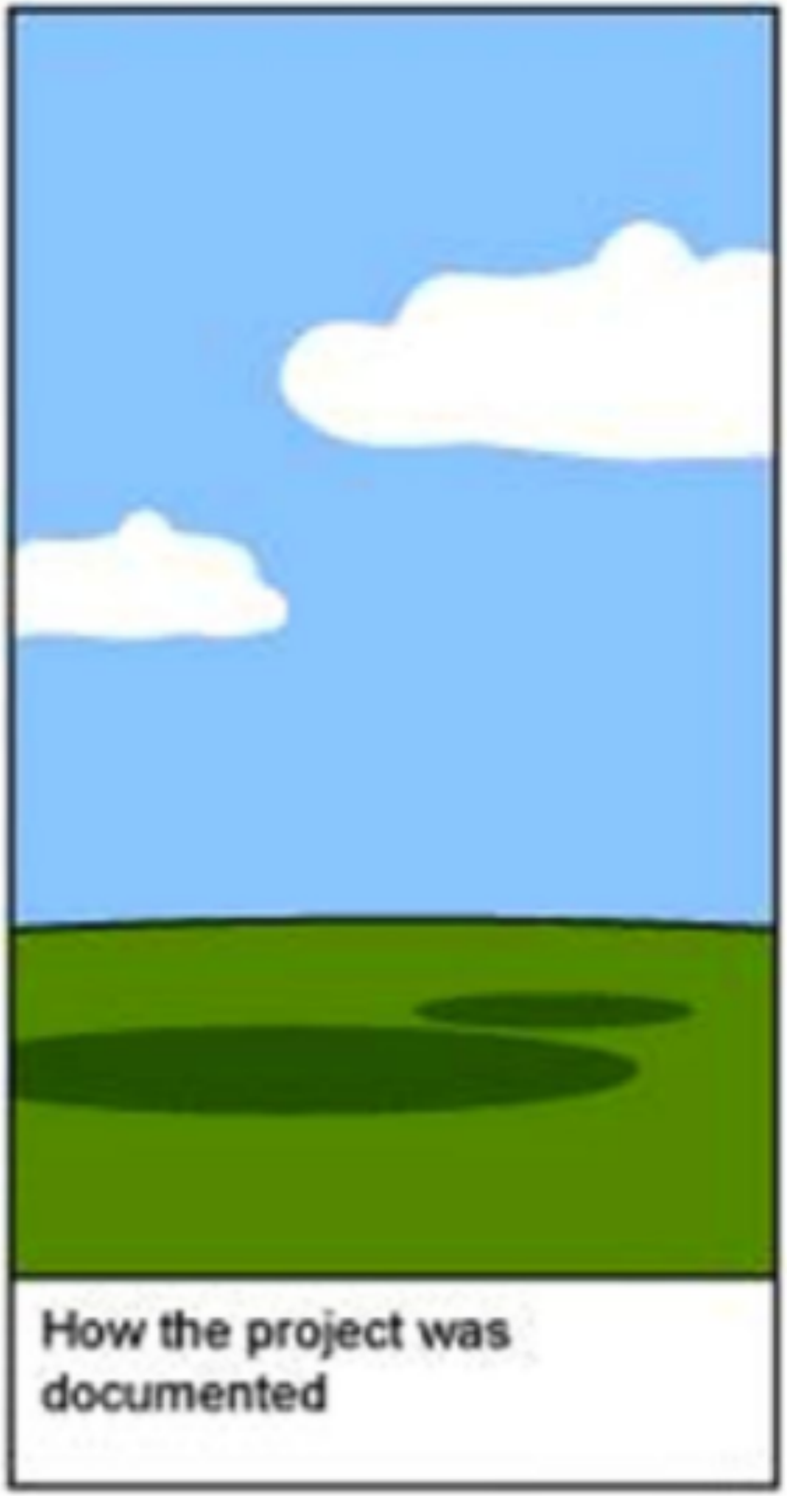
   – External documentation
     - User manual
     - Reference manual
     - Design documents

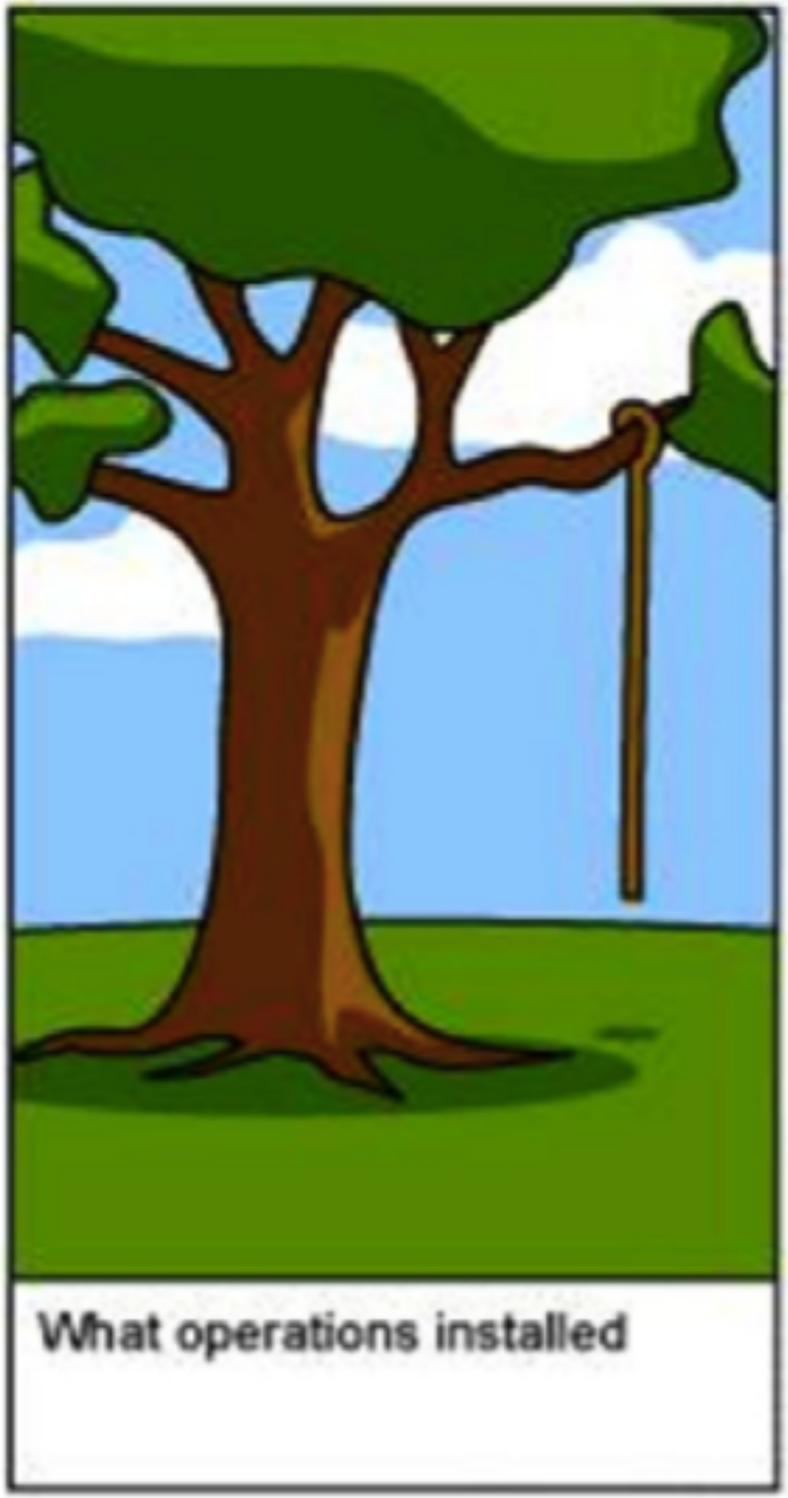

How the customer explained it

How the Project Leader understood it

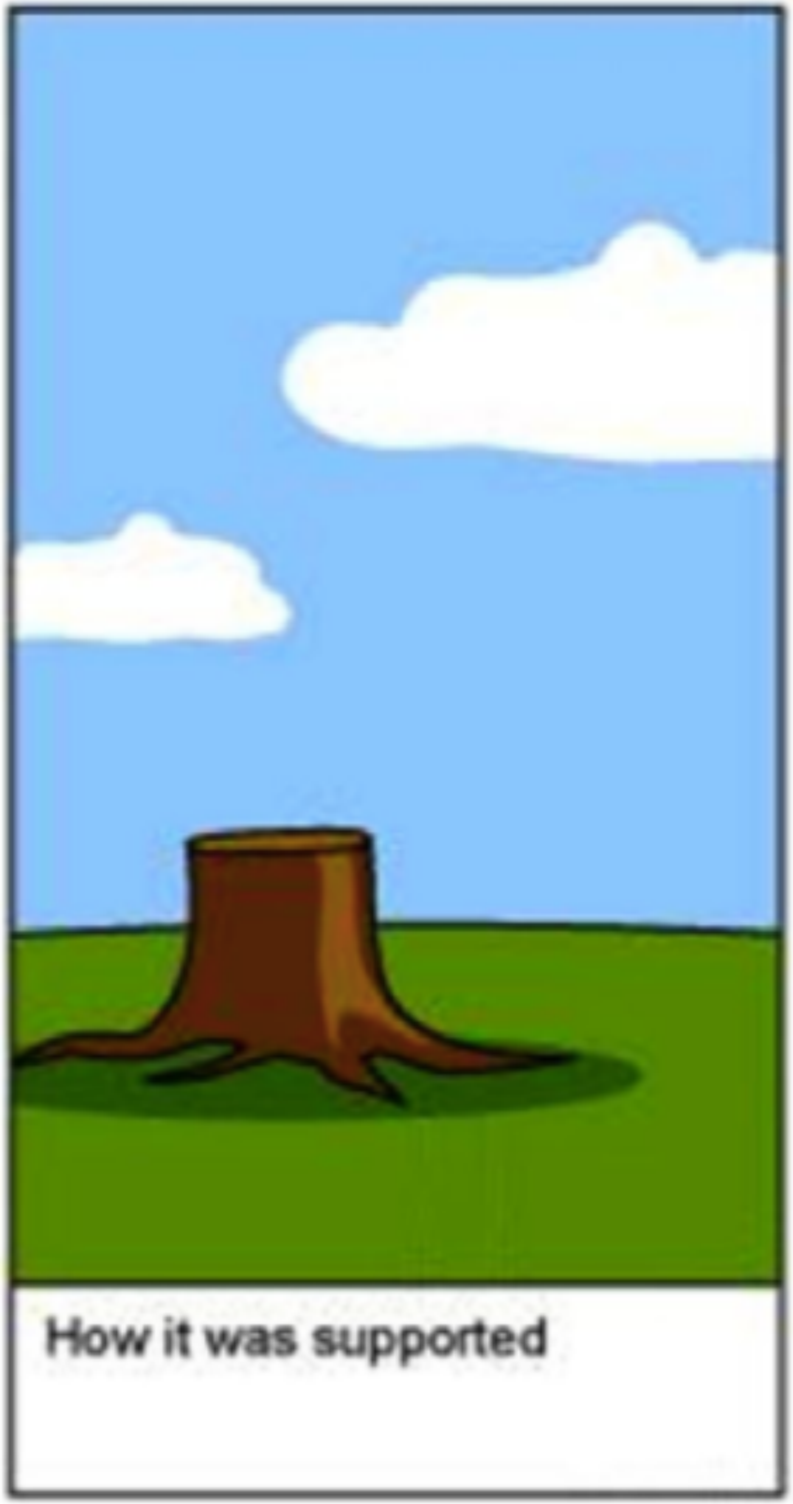
How the Programmer wrote it

How the Business Consultant described it

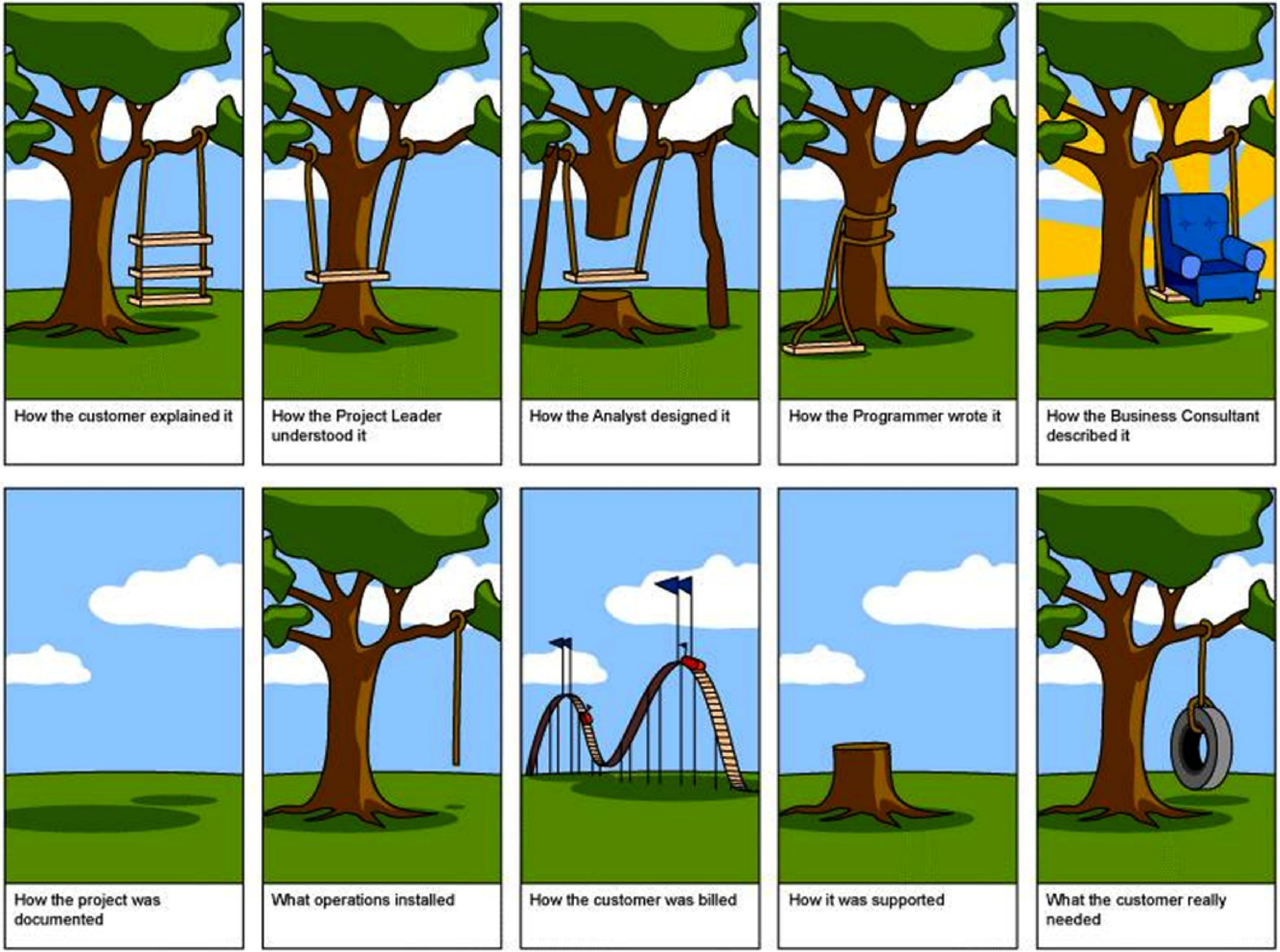

How the project was documented


What operations installed

How the customer was billed


How it was supported

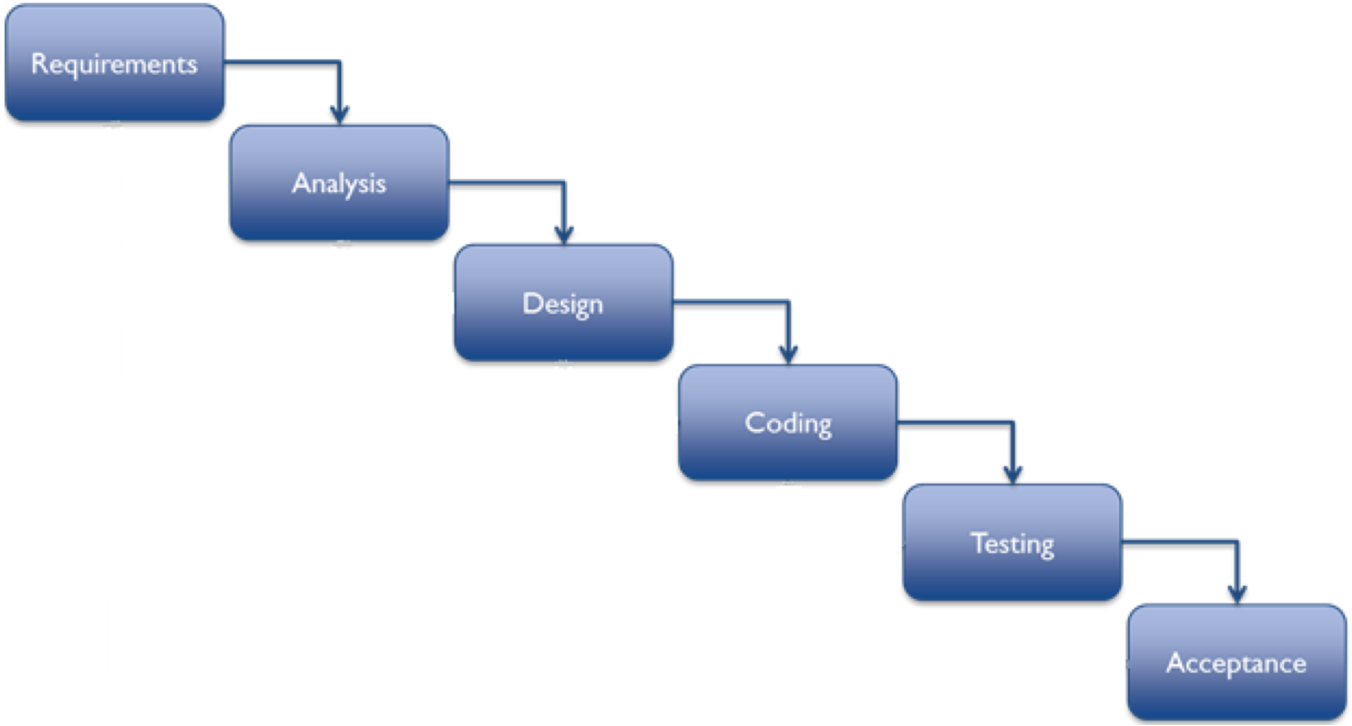
What the customer really needed


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

Software Process Models

- The Waterfall model
  - Separate and distinct phases of specification and development
- Evolutionary development
  - Specification and development are interleaved
- Formal transformation
  - A mathematical system model is formally transformed to an implementation
- Reuse-based development
  - The system is assembled from existing components

Generic Software Process Models



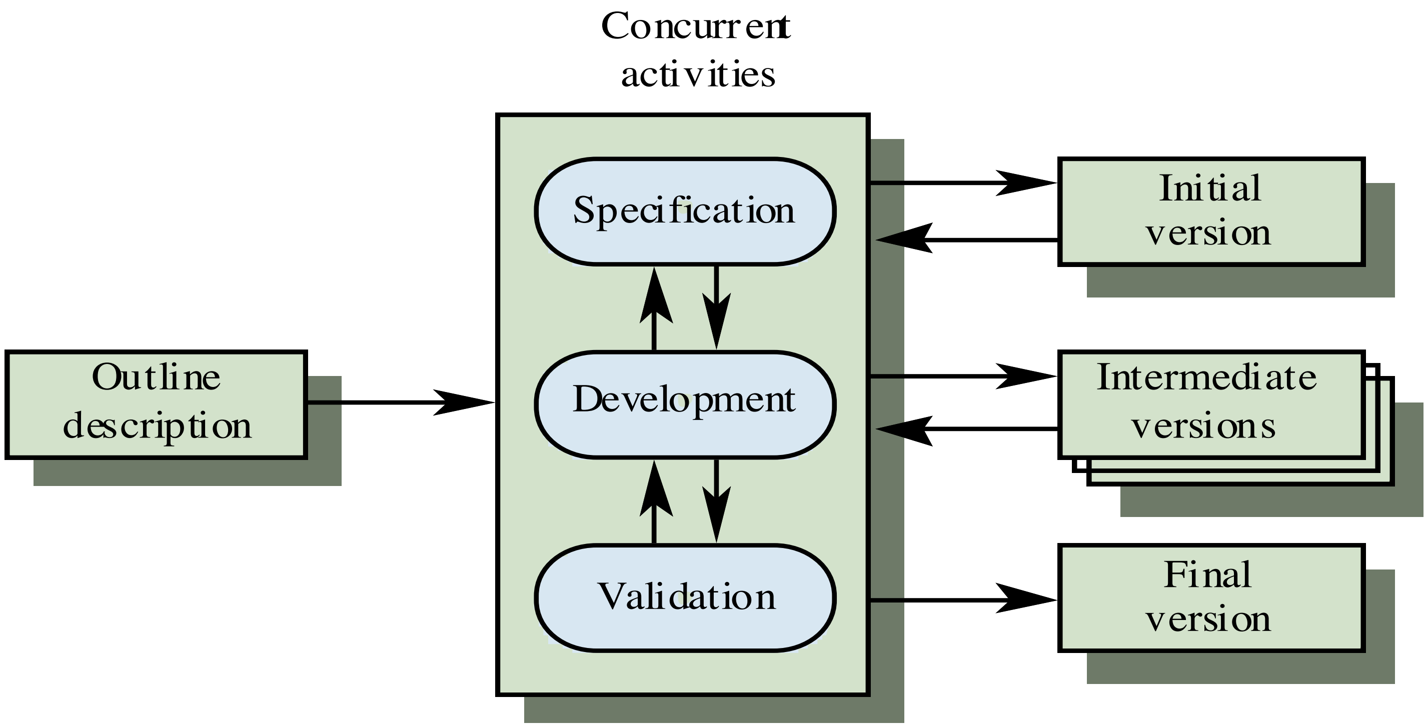
Waterfall Model

Software Process Models

Waterfall Model Phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The drawback of the waterfall model is the difficulty of accommodating change after the process is underway

Software Process Models



Evolutionary Development

Software Process Models

- Exploratory prototyping
  - Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements
- Throw-away prototyping
  - Objective is to understand the system requirements. Should start with poorly understood requirements

Software Process Models

- Problems
  - Lack of process visibility
  - Systems are often poorly structured
  - Special skills (e.g. in languages for rapid prototyping) may be required

- Applicability
  - For small or medium-size interactive systems
  - For parts of large systems (e.g. the user interface)
  - For short-lifetime systems

Software Process Models

Risk Management

- Perhaps the principal task of a engineering manager is to minimise risk
- The 'risk' inherent in an activity is a measure of the uncertainty of the outcome of that activity
- High-risk activities cause schedule and cost overruns
- Risk is related to the amount and quality of available information. The less information, the higher the risk

Software Process Models

Process Model Risk Problems

– Waterfall
  - High risk for new systems because of specification and design problems
  - Low risk for well-understood developments using familiar technology

– Prototyping (Evolutionary)
  - Low risk for new applications because specification and program stay in step
  - High risk because of lack of process visibility

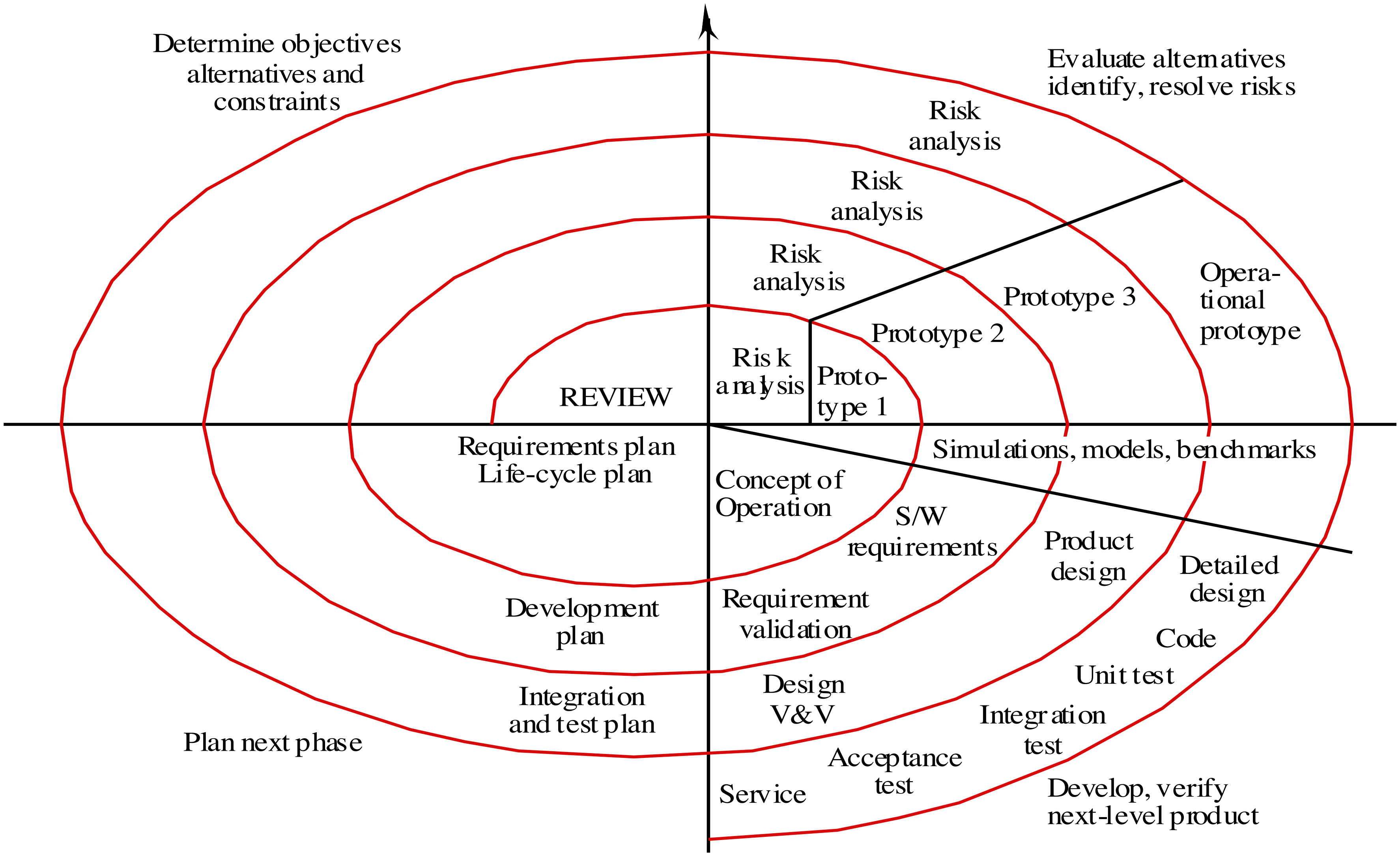
– Transformational
  - High risk because of need for advanced technology and staff skills

Software Process Models

Hybrid Process Models

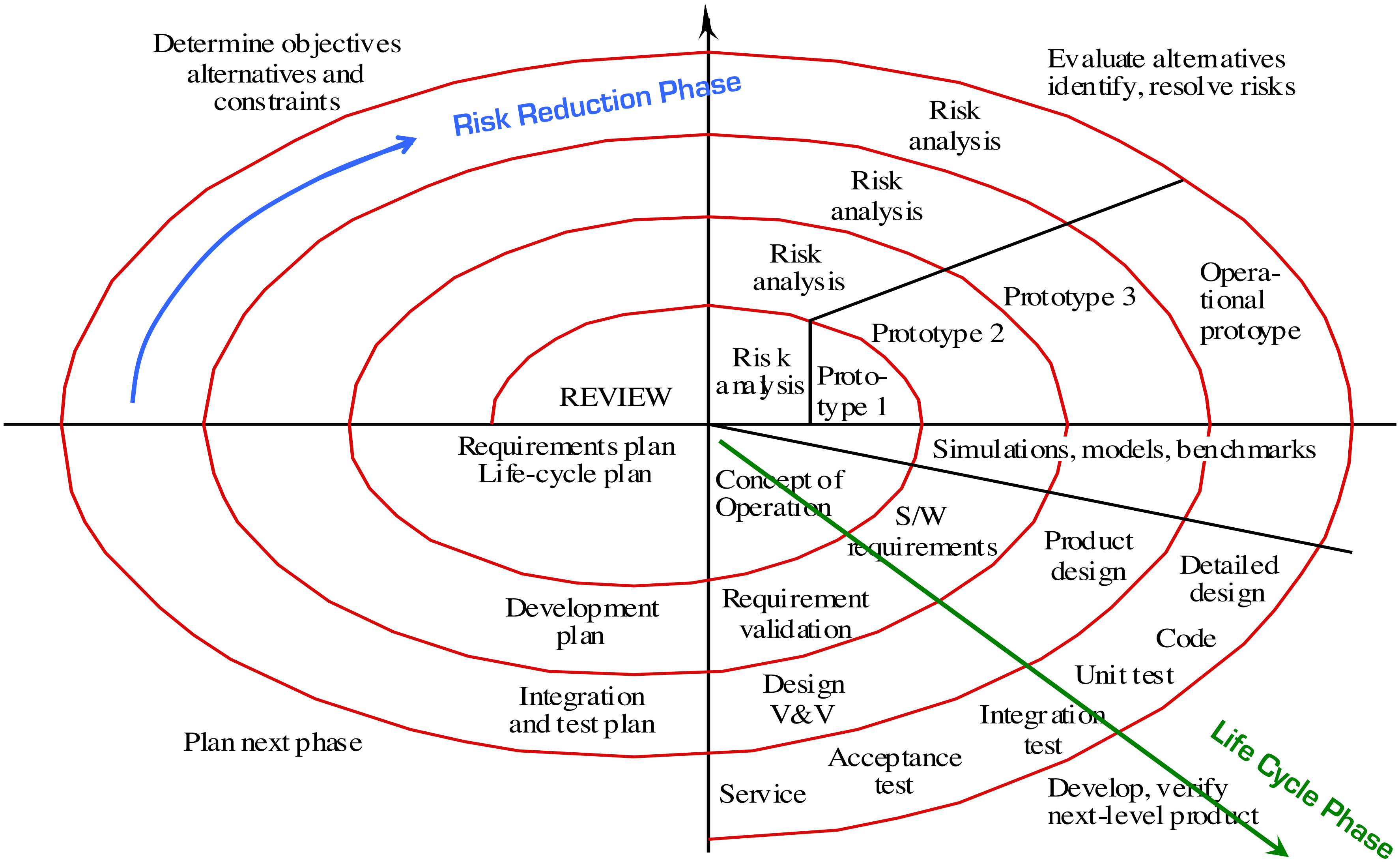
- Large systems are usually made up of several sub-systems
- The same process model need not be used for all subsystems
- Prototyping for high-risk specifications
- Waterfall model for well-understood developments

Software Process Models



Spiral model of the software process

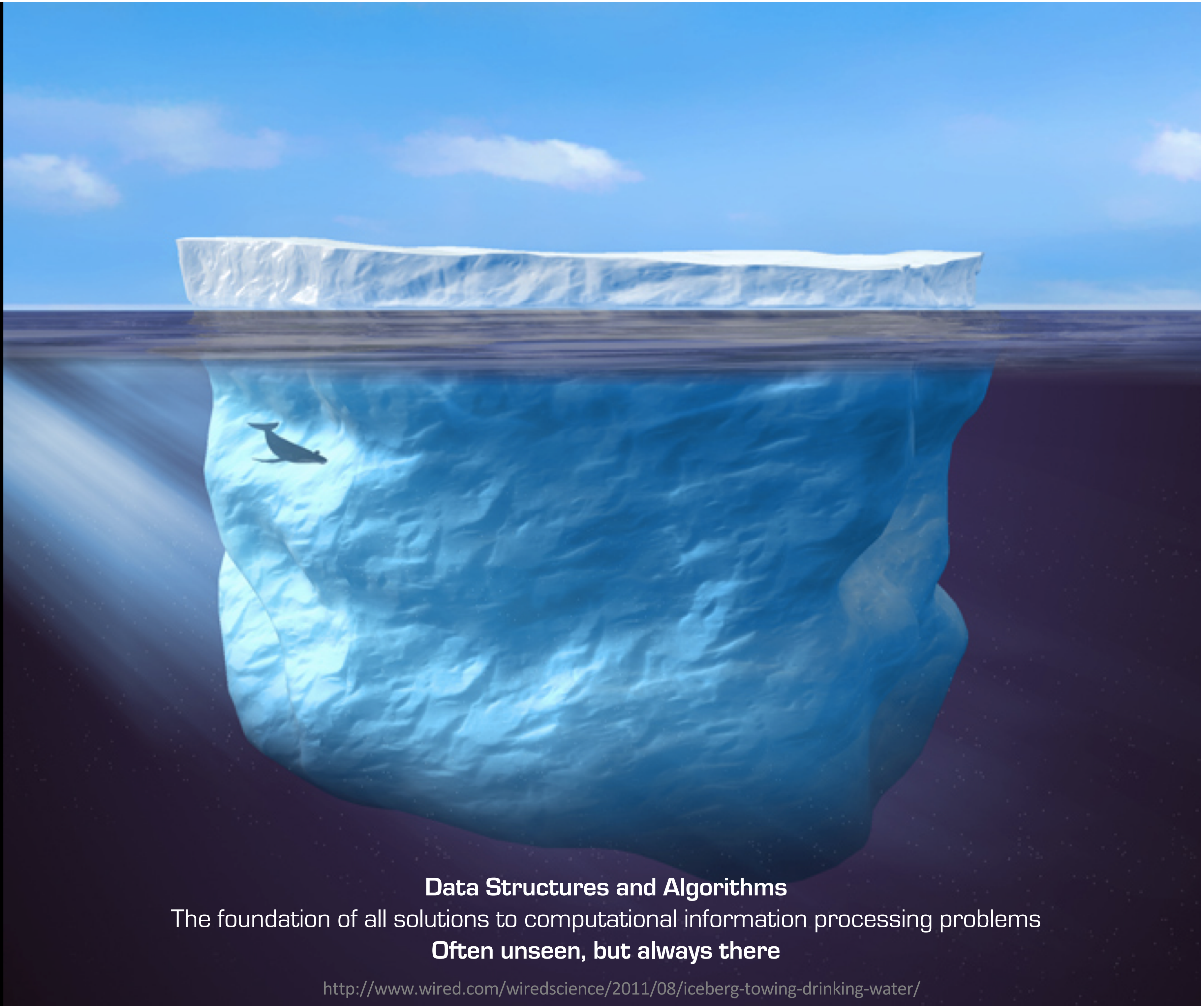
Software Process Models



Software Process Models

Phases of the spiral model

- Objective setting
  - Specific objectives for the project phase are identified
- Risk assessment and reduction
  - Key risks are identified, analysed and information is sought to reduce these risks
- Development and validation
  - An appropriate model is chosen for the next phase of development
- Planning
  - The project is reviewed and plans drawn up for the next round of the spiral


Data Structures and Algorithms
The foundation of all solutions to computational information processing problems
Often unseen, but always there
http://www.wired.com/wiredscience/2011/08/iceberg-towing-drinking-water/