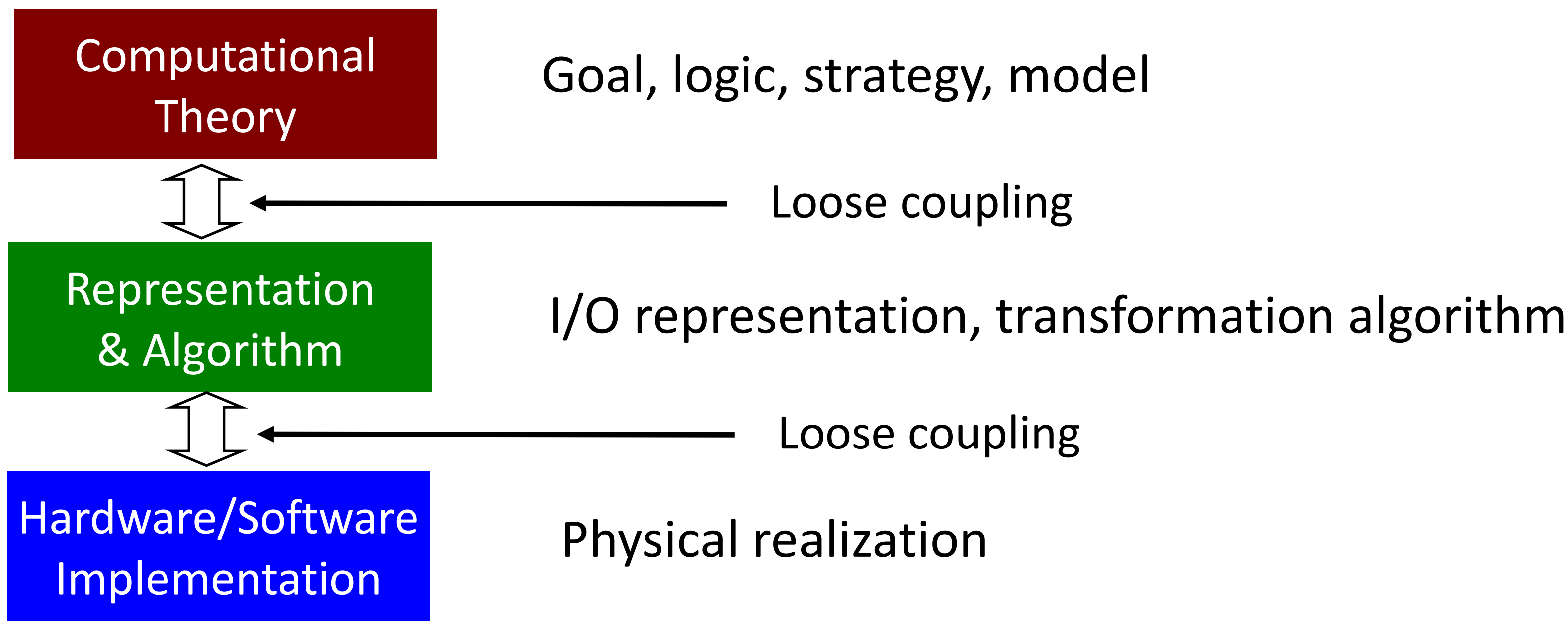
Software Development

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

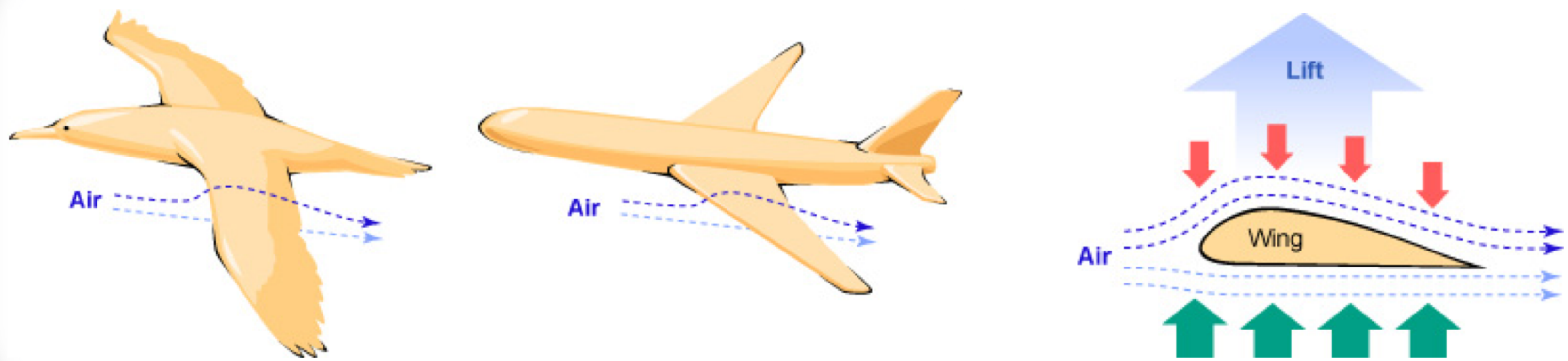
Marr's Hierarchy of Abstraction / Levels of Understanding Framework

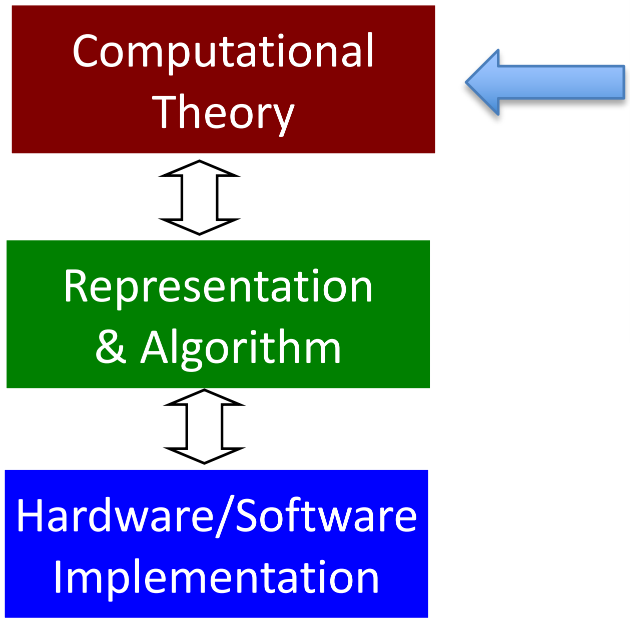


Marr's Hierarchy of Abstraction / Levels of Understanding Framework

"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.



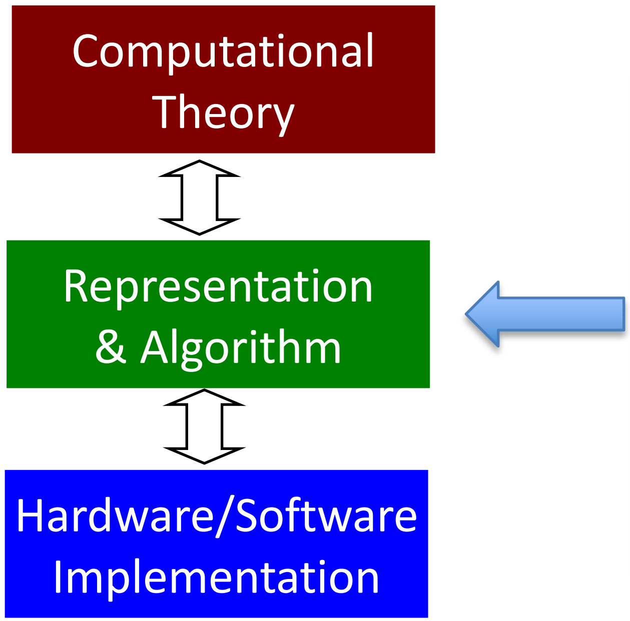


Sorting a List

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

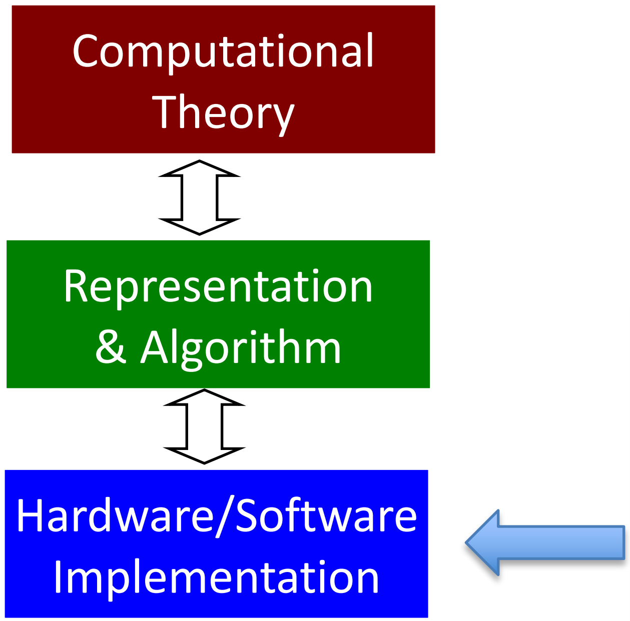
Bubble Sort

Insertion Sort

Quick Sort

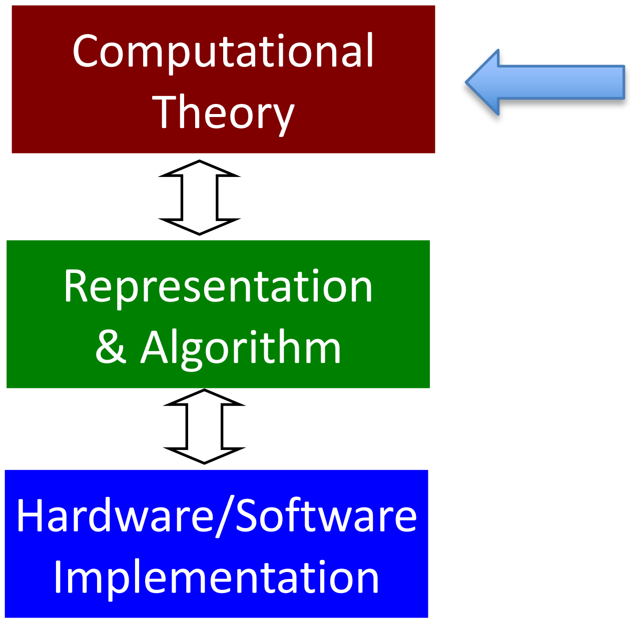
Merge Sort, …

Key point: different computational efficiency



Sorting a List

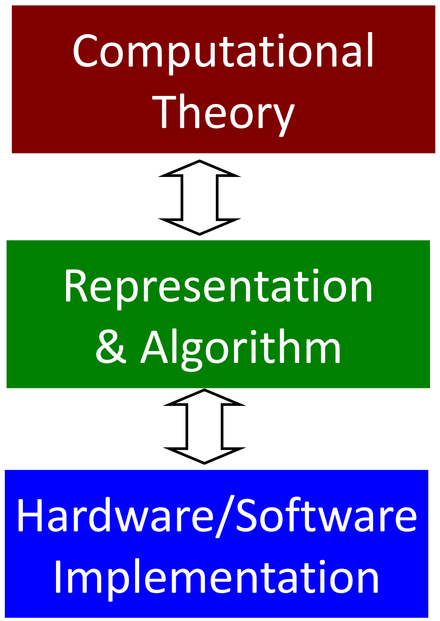
```
insertion_sort(item s[], int n)
{
        int i,j;                /* counters */

        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;
                }
        }
}
```



Fourier Transform

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

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



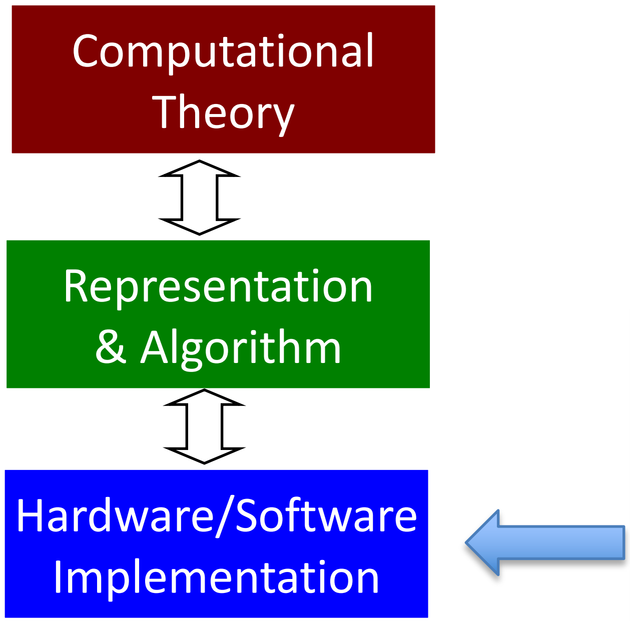
Fourier Transform

DFT: Discrete Fourier Transform

FFT: Fast Fourier Transform

FFTW: Fasted Fourier Transform in the West

Key point: different computational efficiency



Fourier Transform

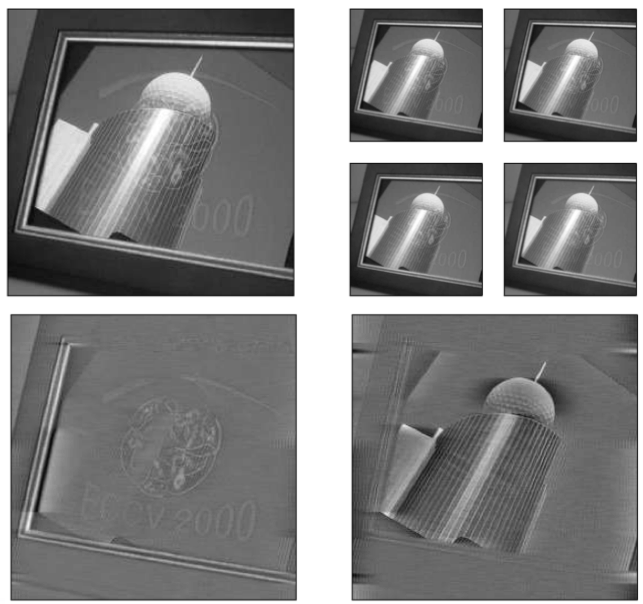
```
main()
{
        unsigned long i;
        int isign;
        float *data1,*data2,*fft1,*fft2;

        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;
}
```

Computational Theory

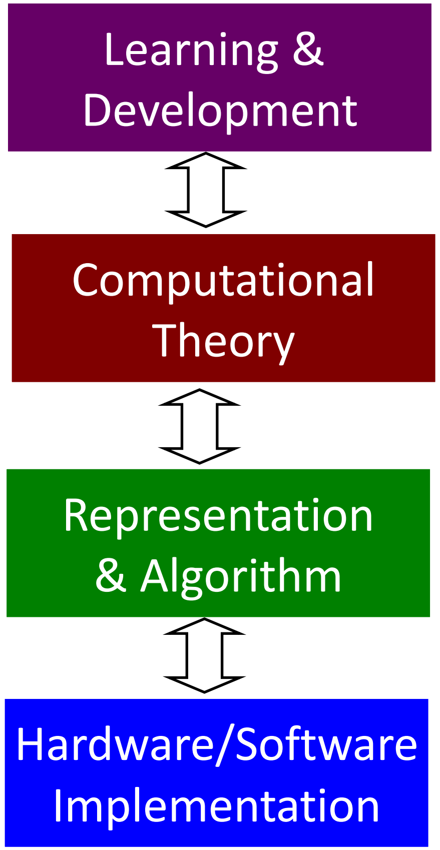
Representation & Algorithm

Hardware/Software Implementation

Fourier Transform

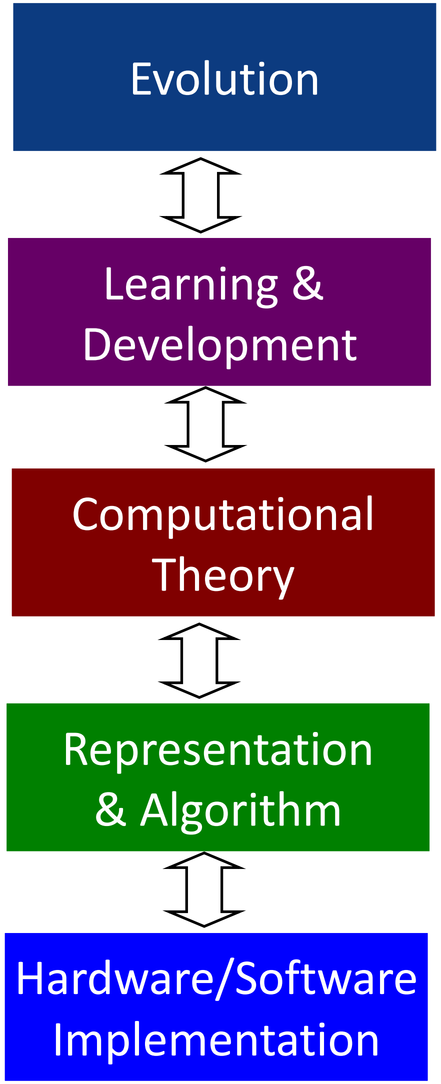


Marr's Levels of Understanding Framework updated 2012 by T. Poggio



Calibrating & improving the model

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

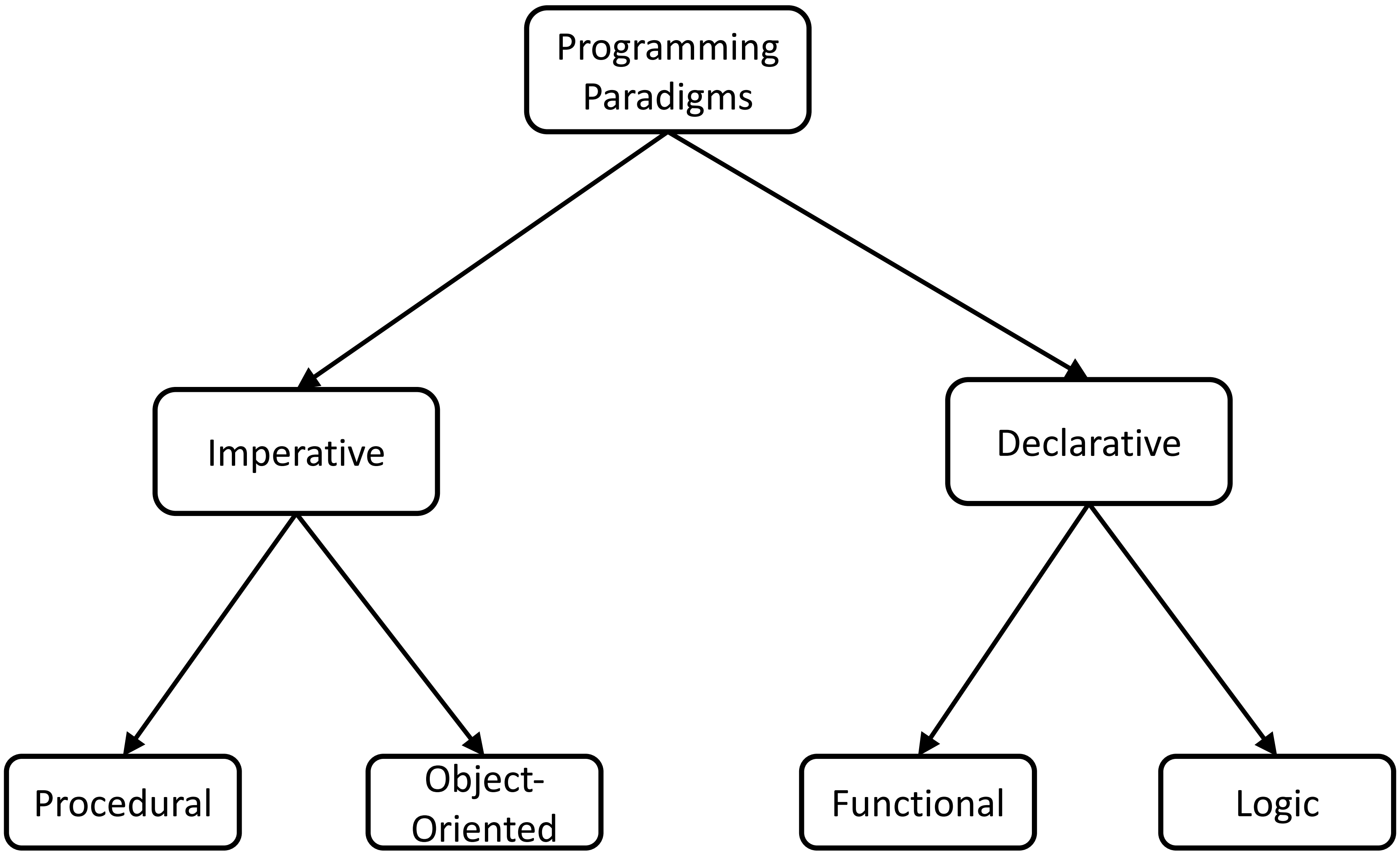


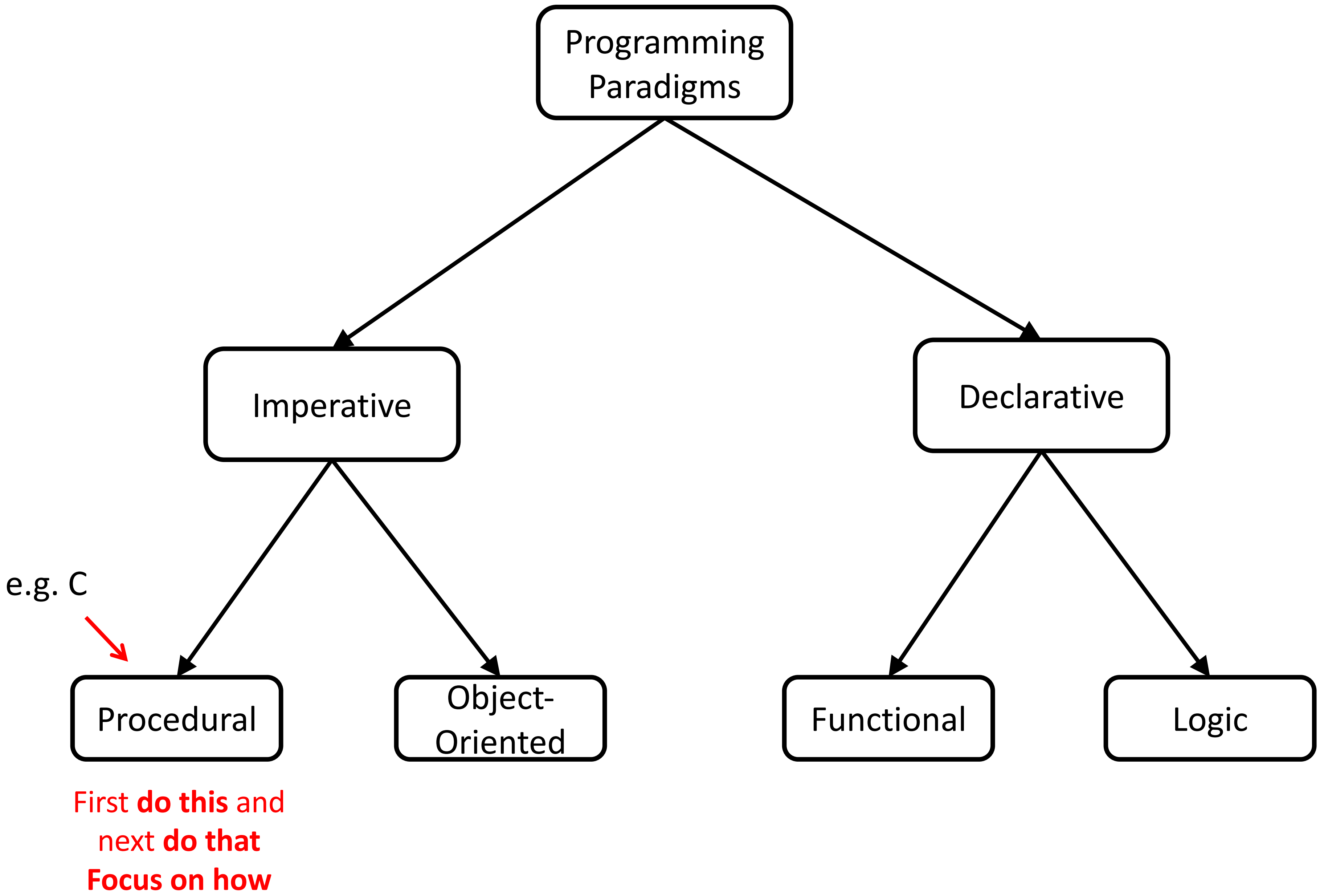
Generating new models

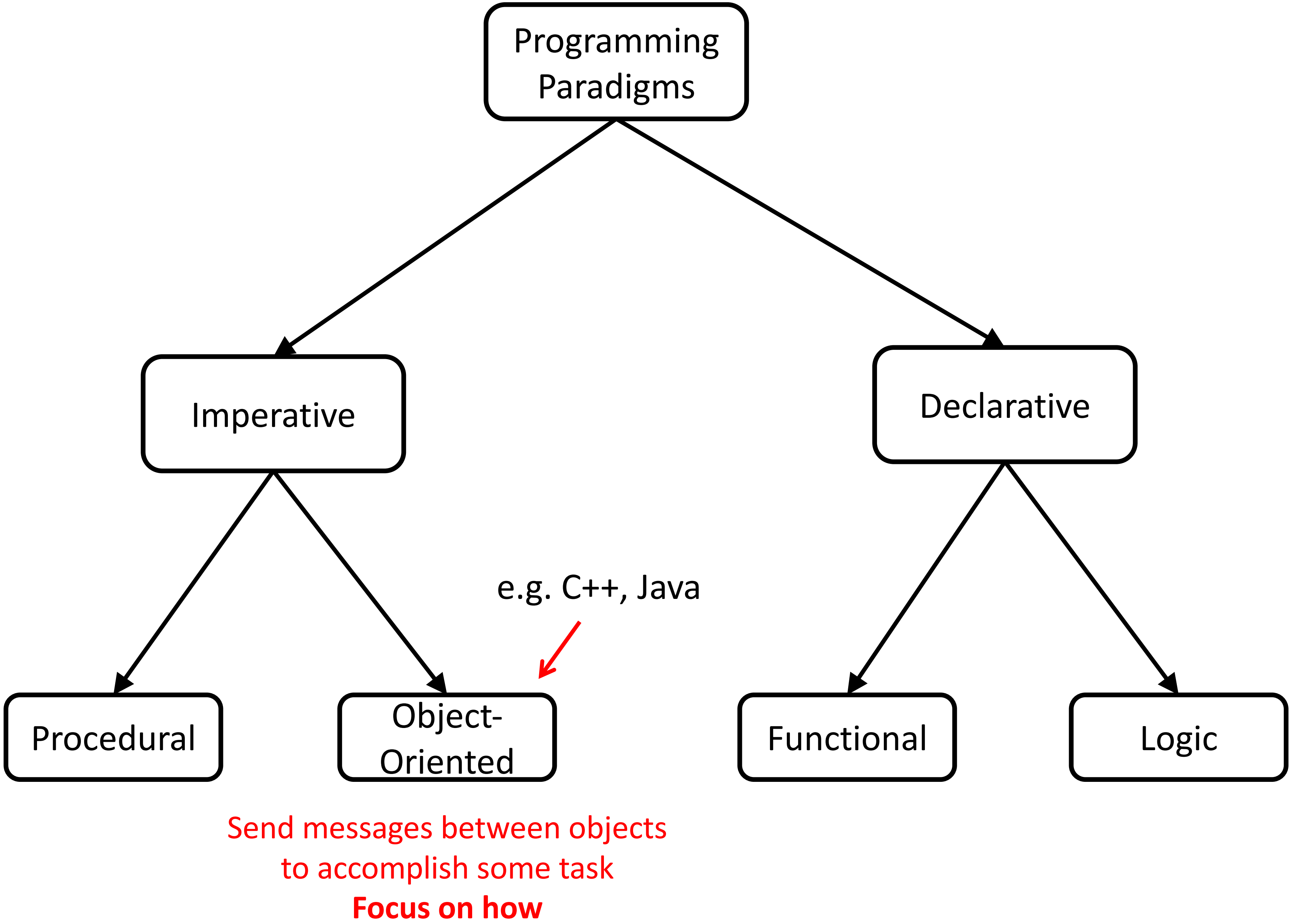
Calibrating & improving the model

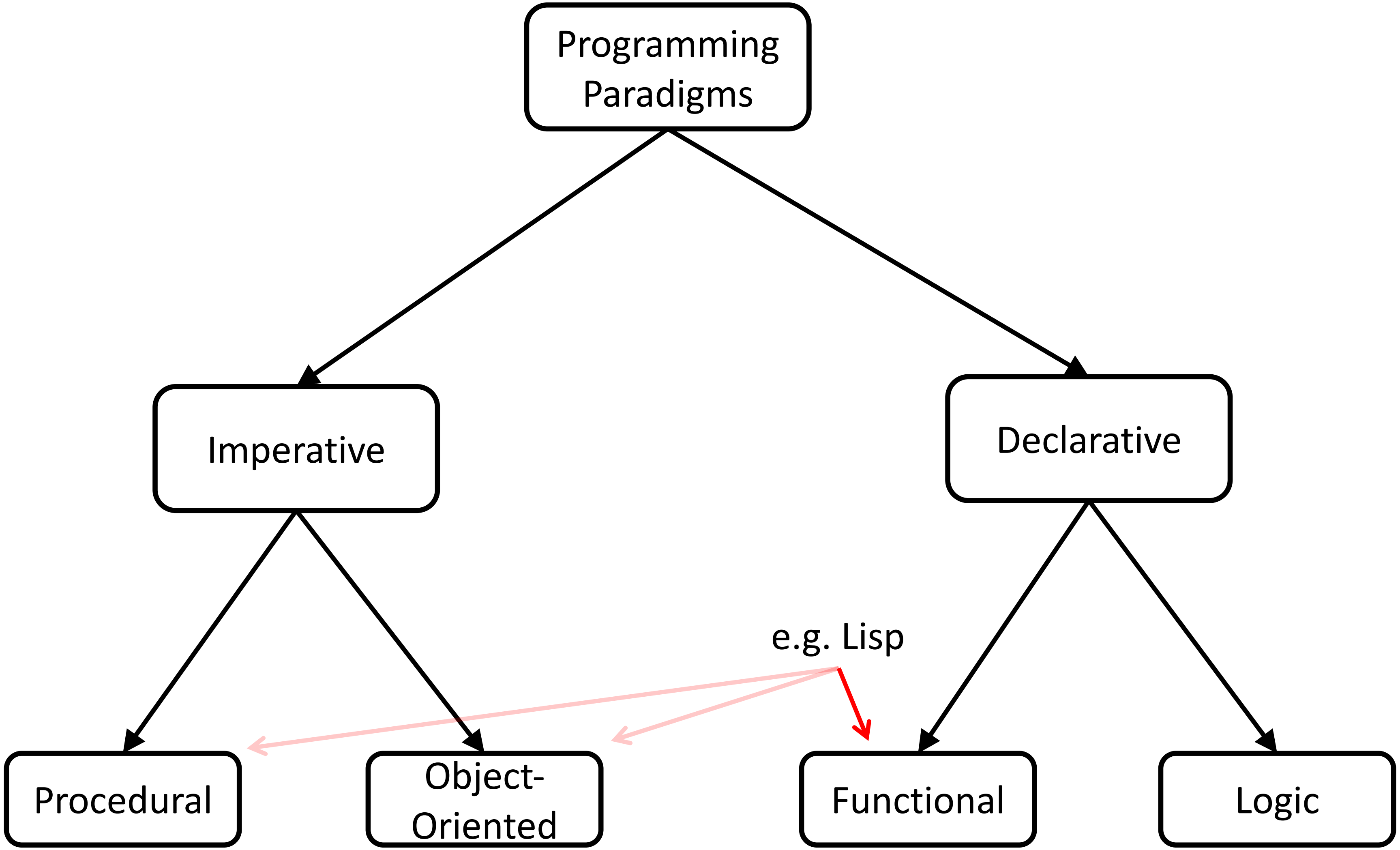
Programming Paradigms

Note: This is an oversimplification

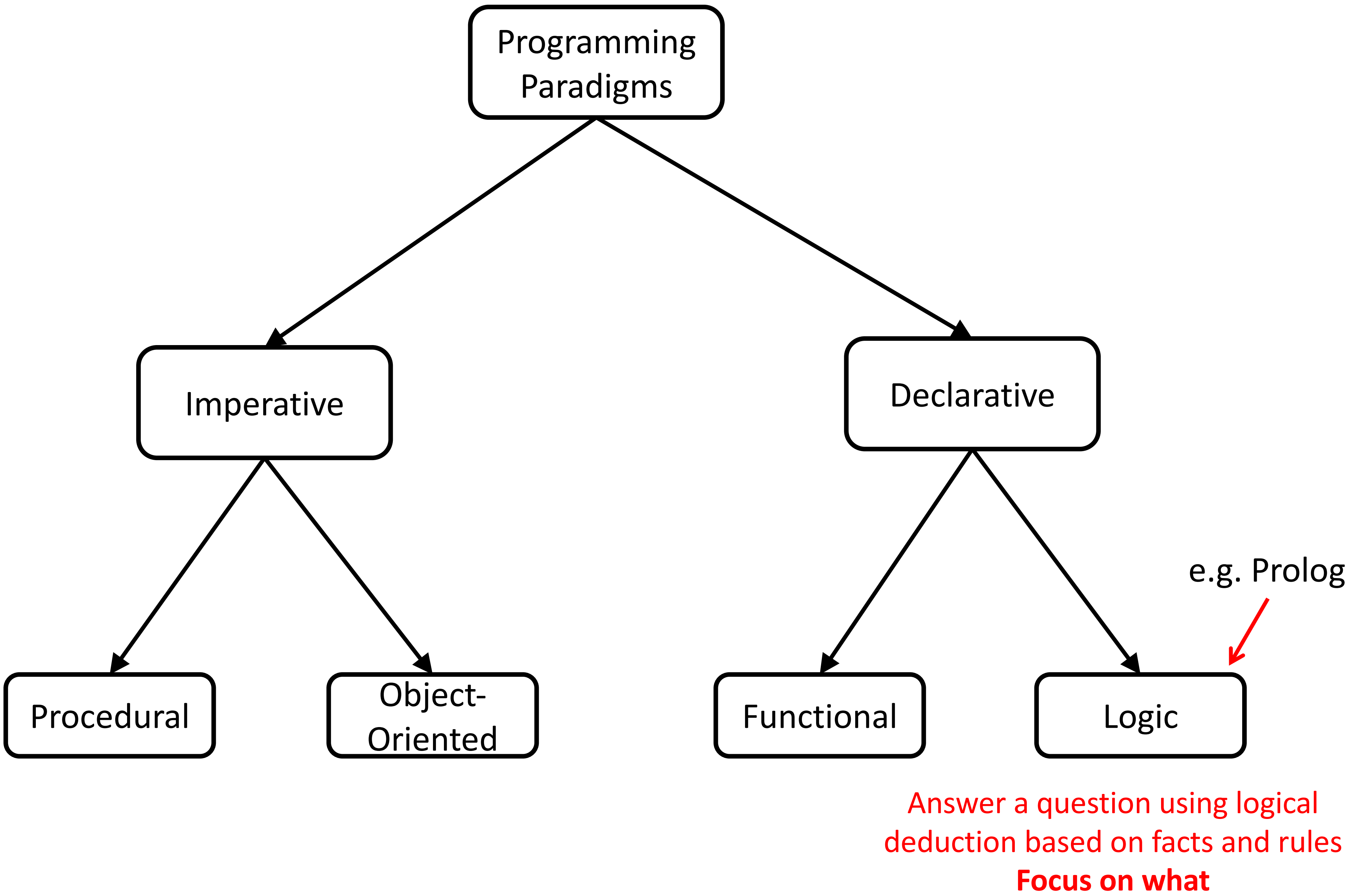


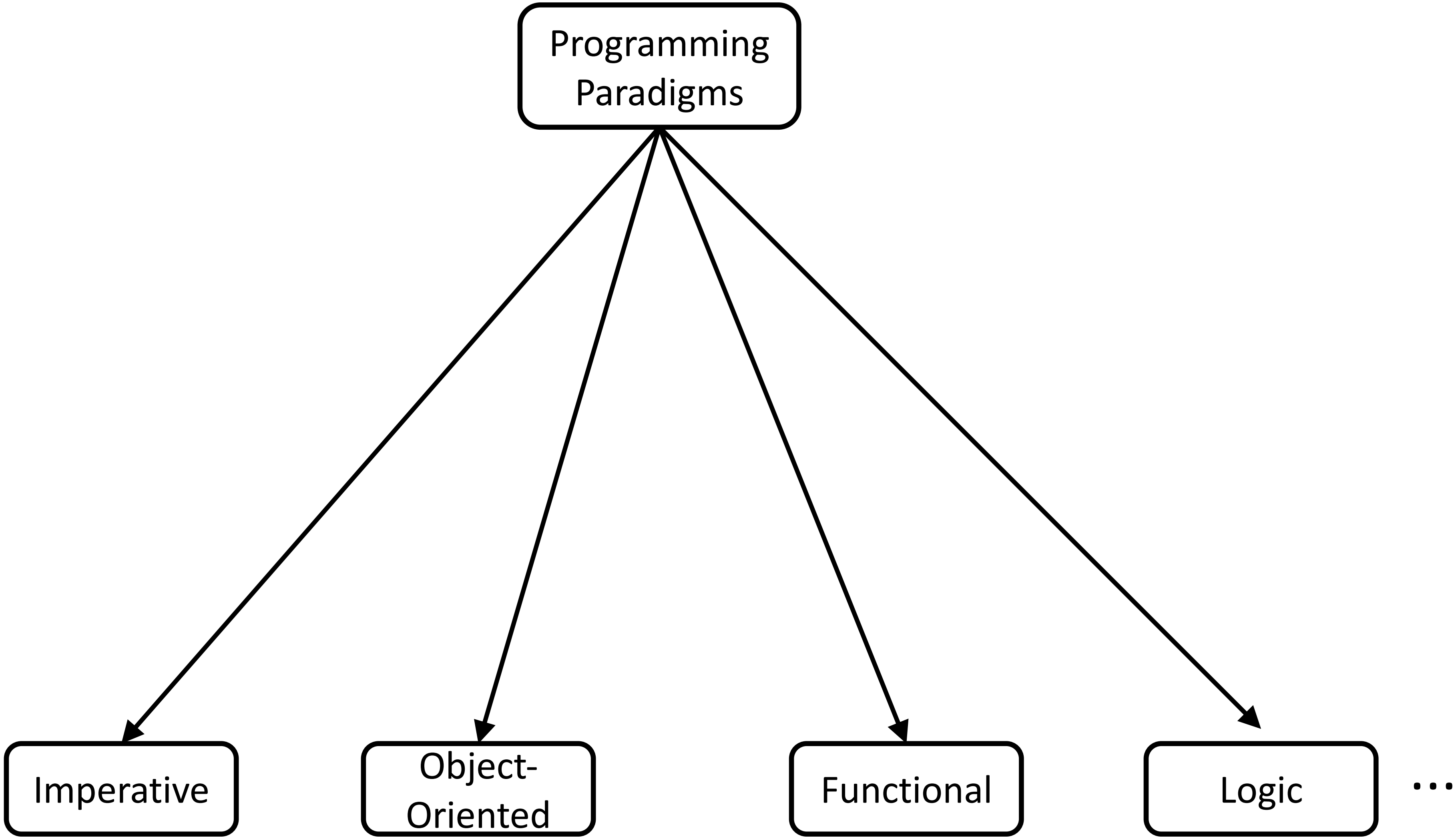

Programming Paradigms
Imperative
Declarative
e.g. C
Procedural
Object-Oriented
Functional
Logic
First **do this** and next **do that**
Focus on how


Programming Paradigms
Imperative
Declarative
Procedural
Object-Oriented
Functional
Logic
e.g. C++, Java
Send messages between objects
to accomplish some task
Focus on how



Evaluate an expression and use the resulting value for something
Focus on what


Programming Paradigms
Imperative
Declarative
Procedural
Object-Oriented
Functional
Logic
e.g. Prolog
Answer a question using logical deduction based on facts and rules
Focus on what



http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html

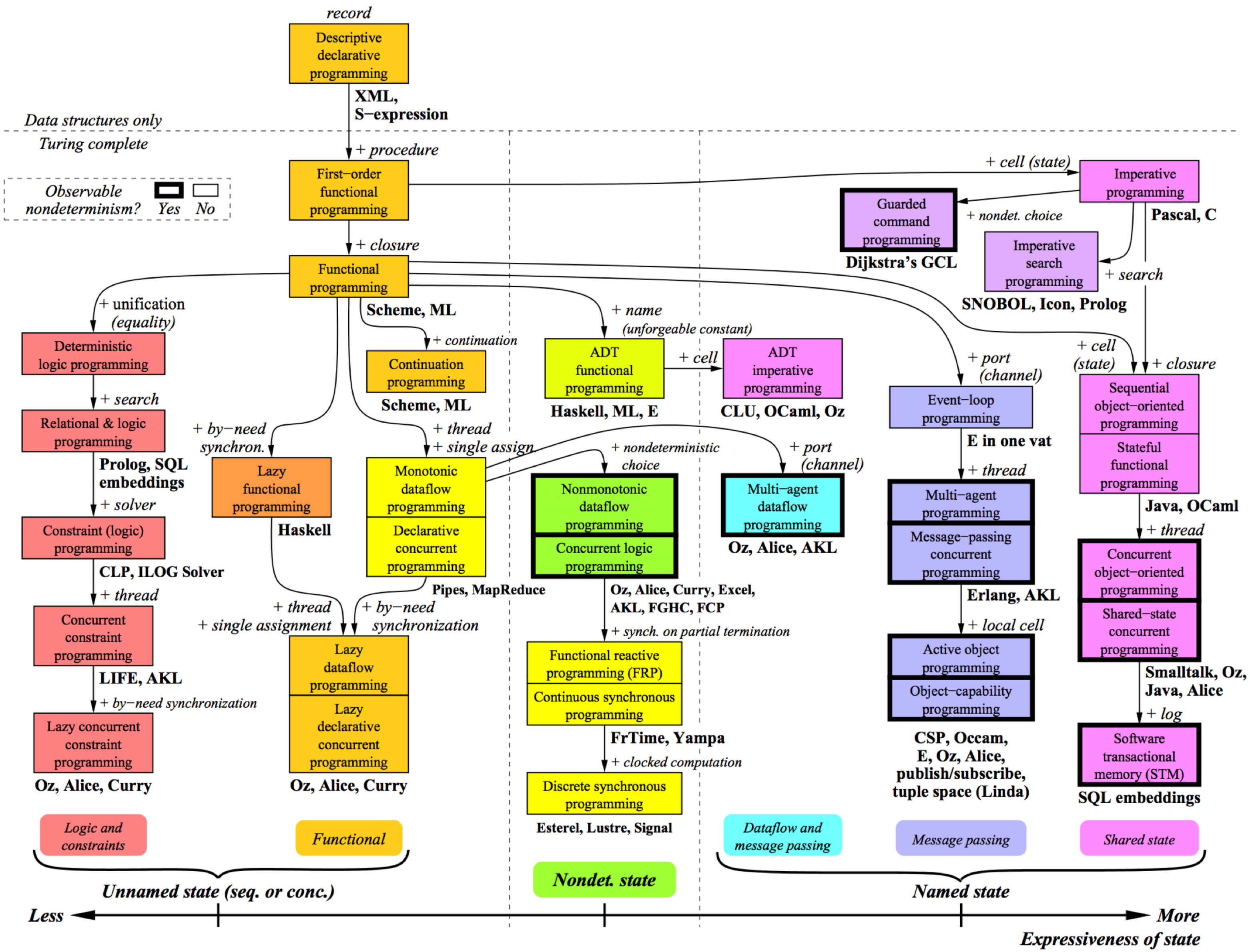


Figure 2. Taxonomy of programming paradigms

Credit: Peter van Roy https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

Comment

The book discussed in this paper is available here.

Paper

PDF

Credit: Shriram Krishnamurthi http://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/

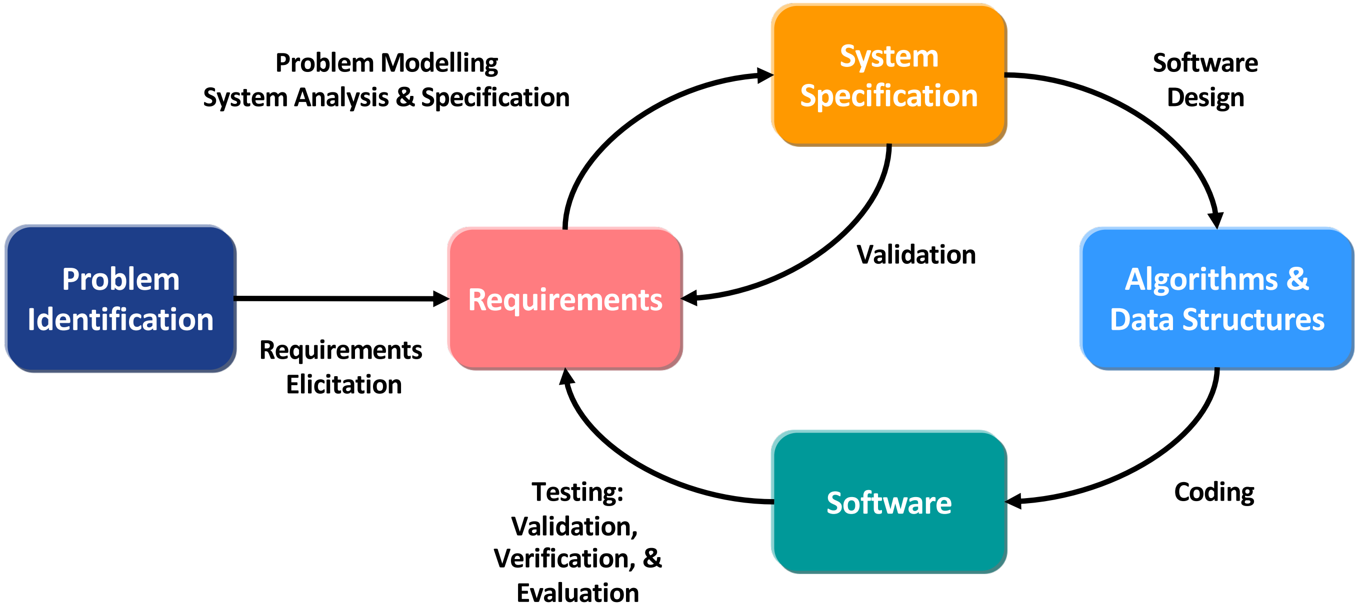
Programming Paradigms:

Ways of thinking or looking at a problem

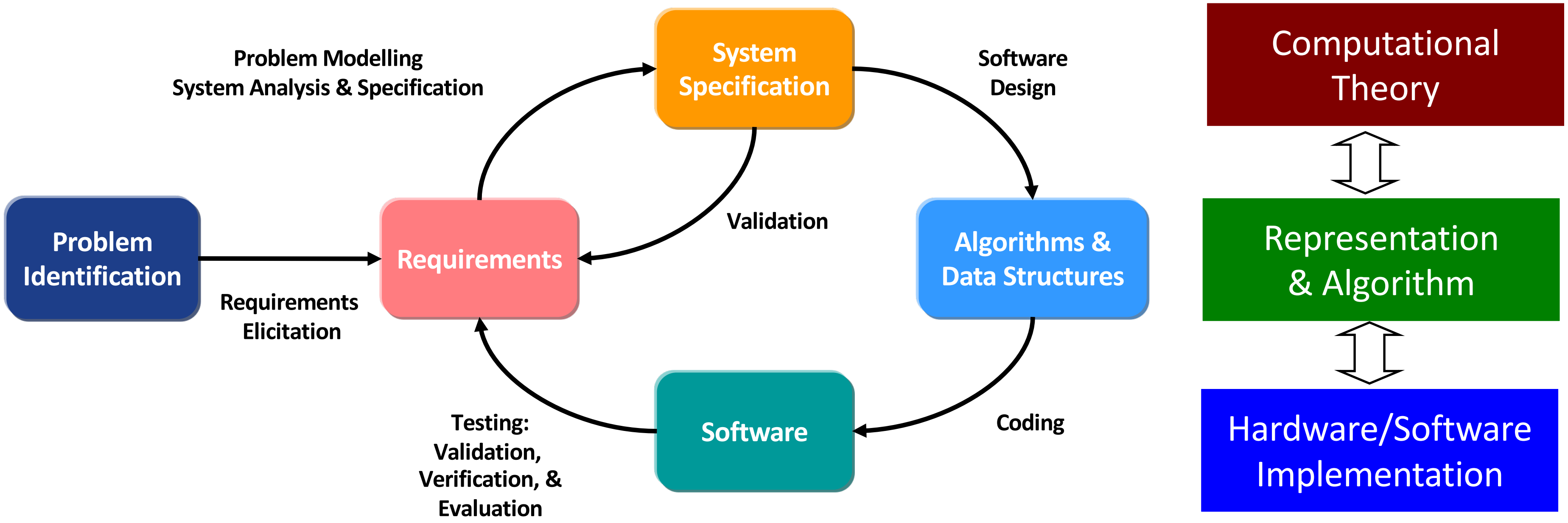
(not so useful as a way of classifying languages)

The Software Development Life Cycle

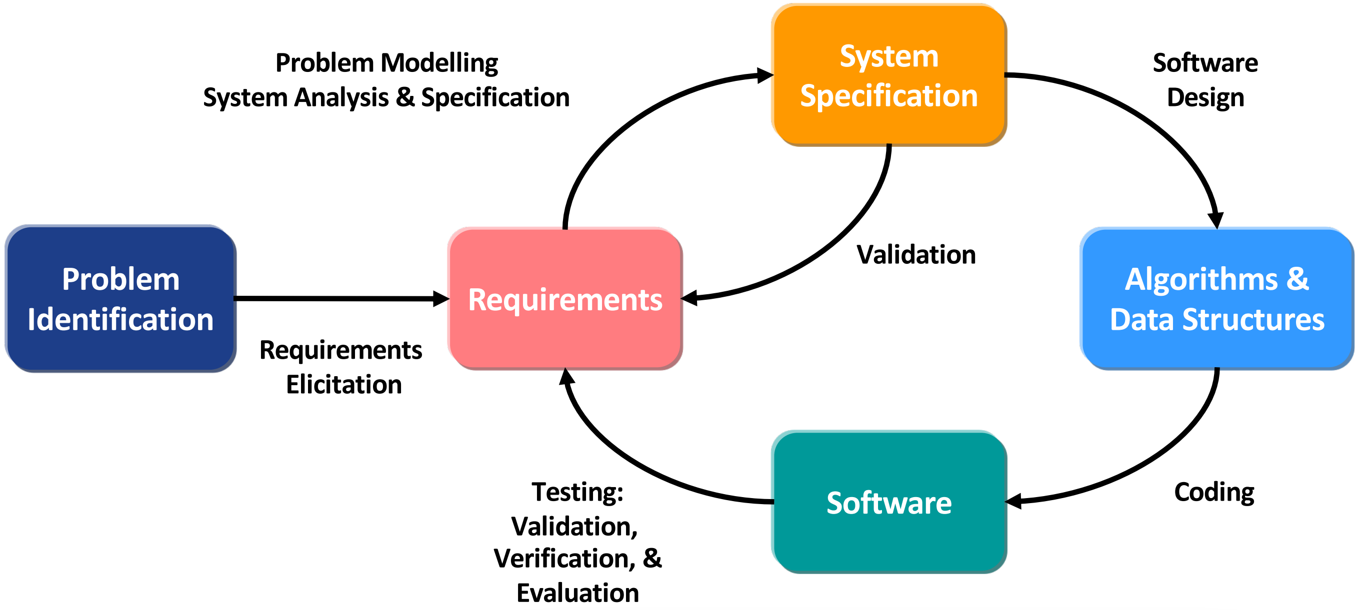
The Software Development Life Cycle

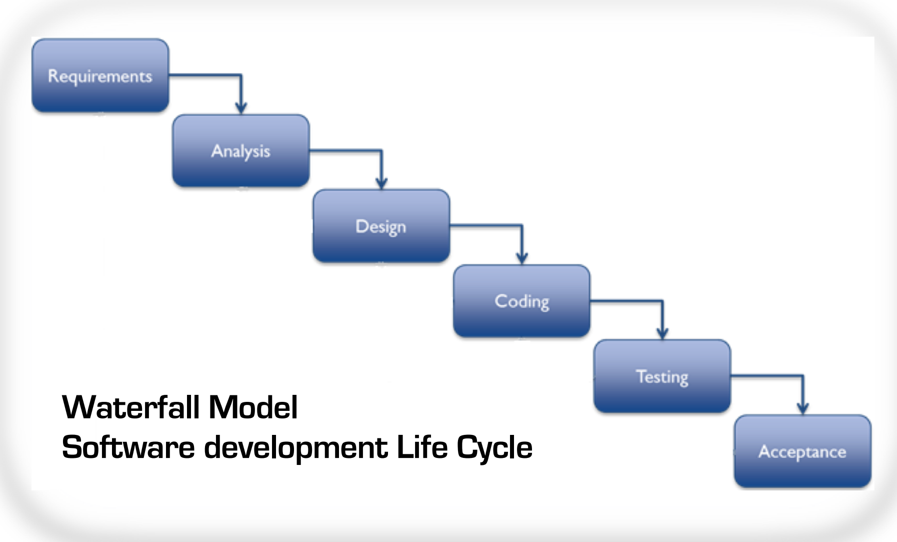


The Software Development Life Cycle

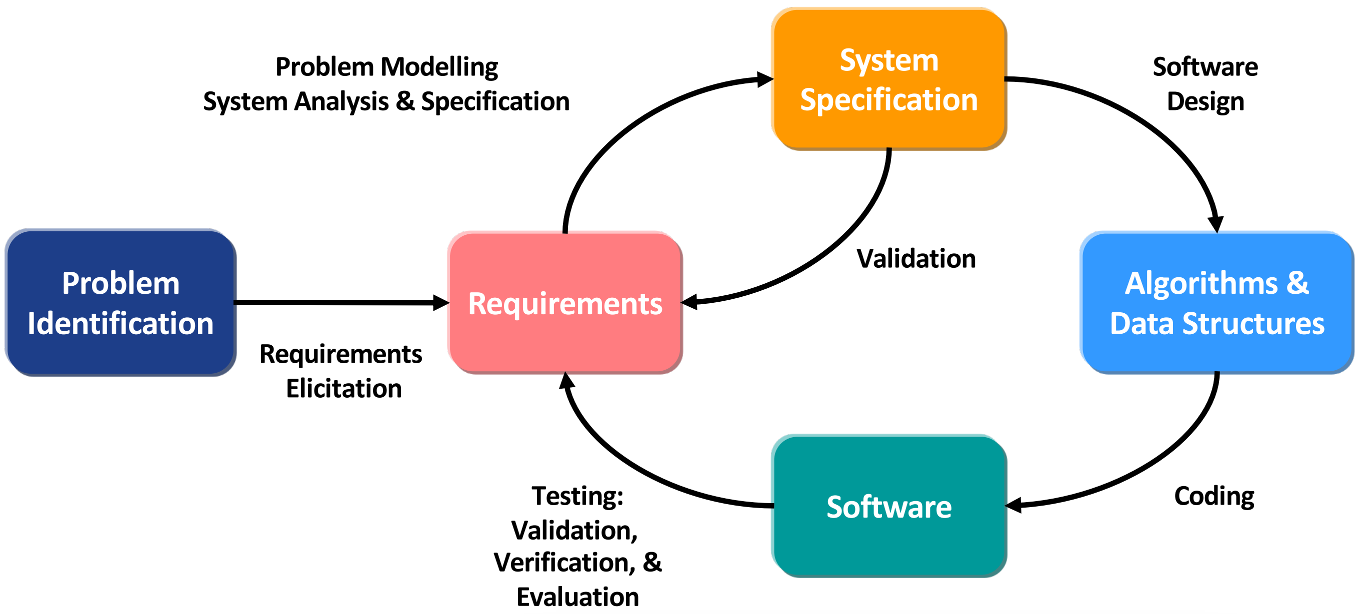


The Software Development Life Cycle





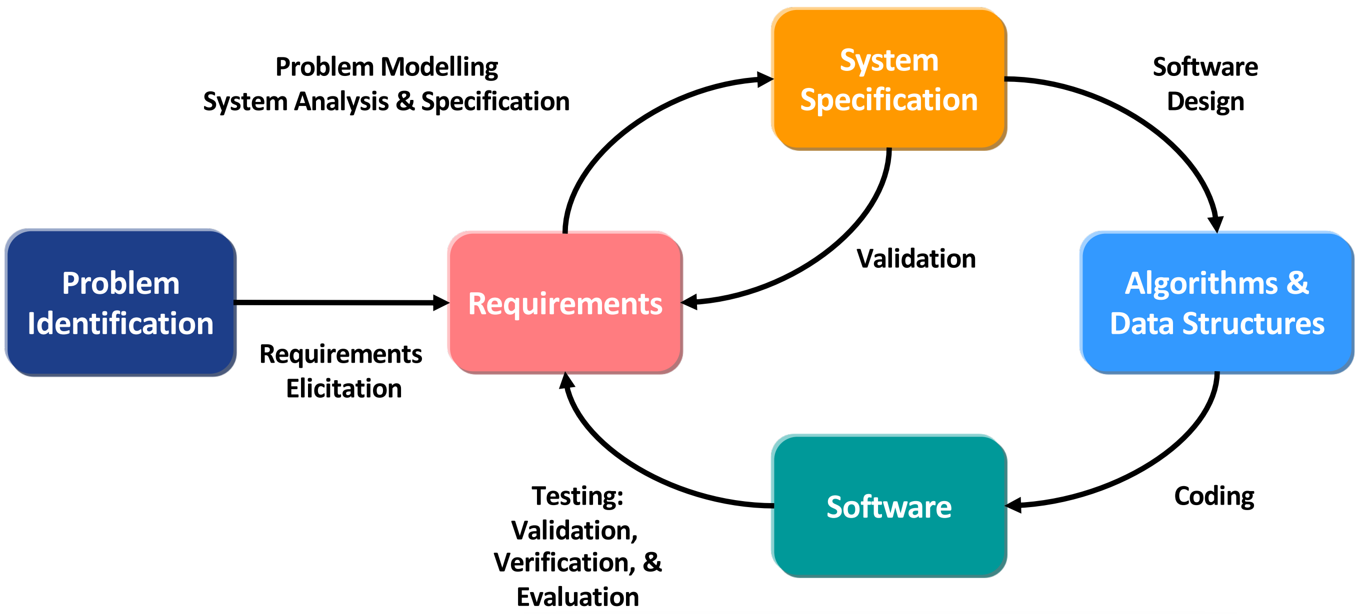
The Software Development Life Cycle



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

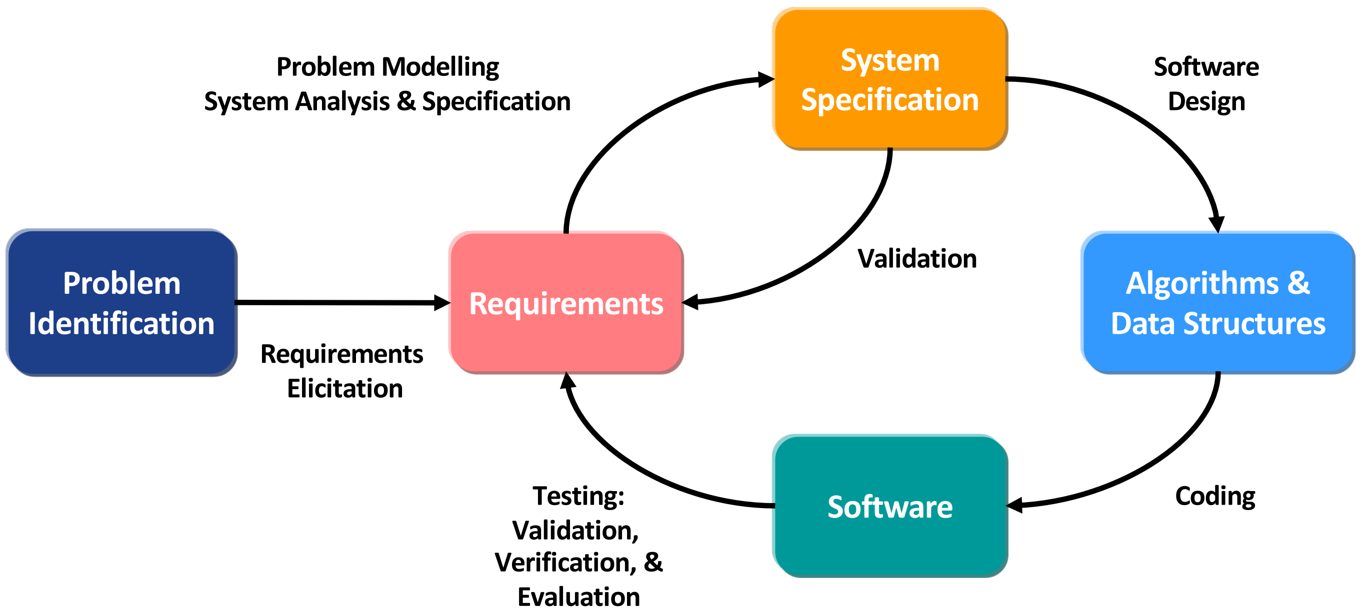
- Yourdon Structured Analysis (YSA)
- Jackson Structured Analysis (JSA)
- Structured Analysis and Design Technique (SADT)

Object-oriented analysis, design, programming
Component-based software engineering (CBSE)

Procedural

Object-oriented

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)

Views a system from the perspective of the data flowing through it and the processes that transform that data

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

Computational Theory

Representation & Algorithm

Hardware/Software Implementation

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

Different methodologies for different paradigms

Procedural Paradigm

Structured Analysis, Design, and Test

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 Engineering: A Practitioner's Approach

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
     - 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

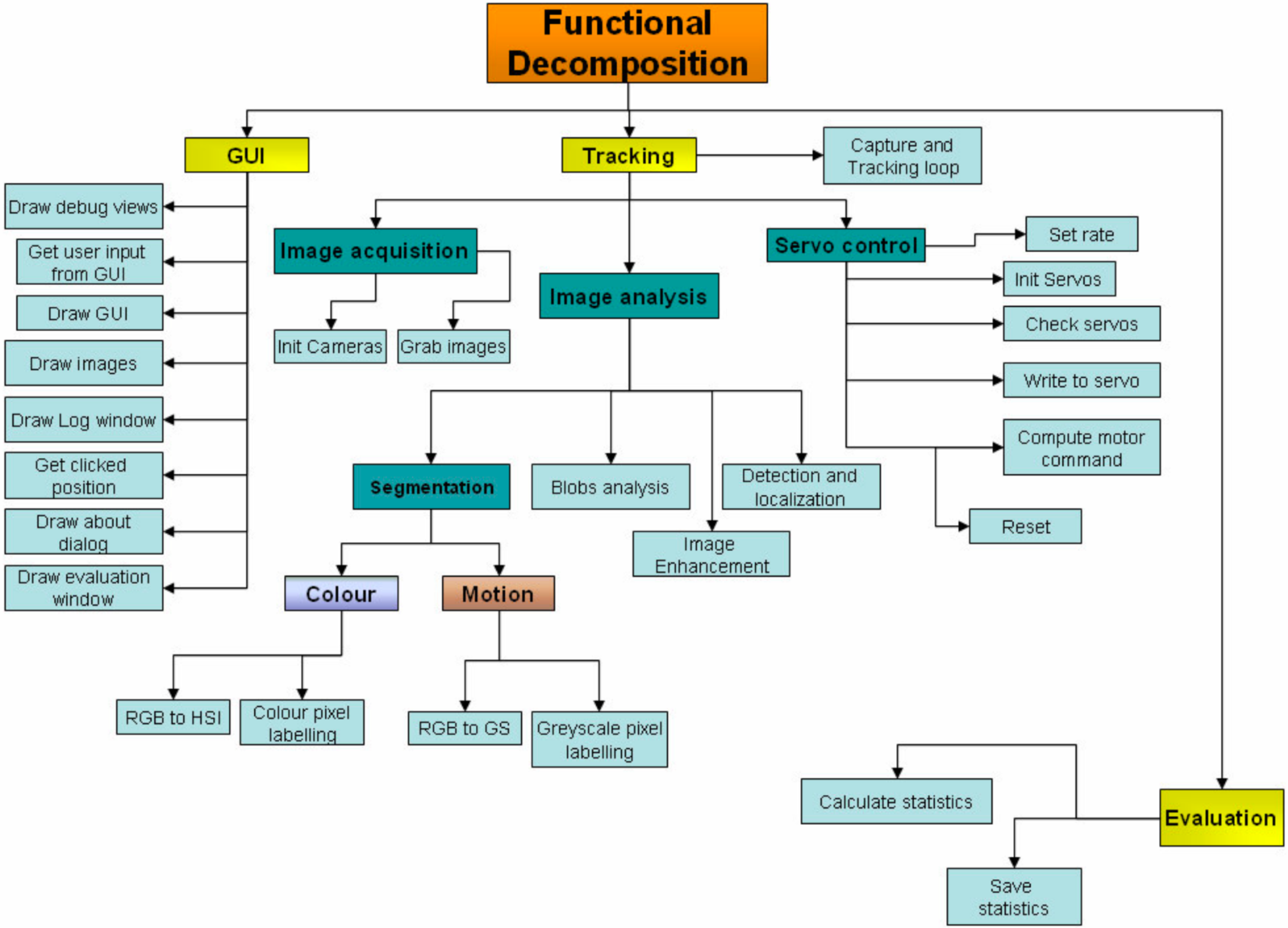
   – Including
     - Functional model
     - Data model
     - Process-flow model
     - Behavioural model

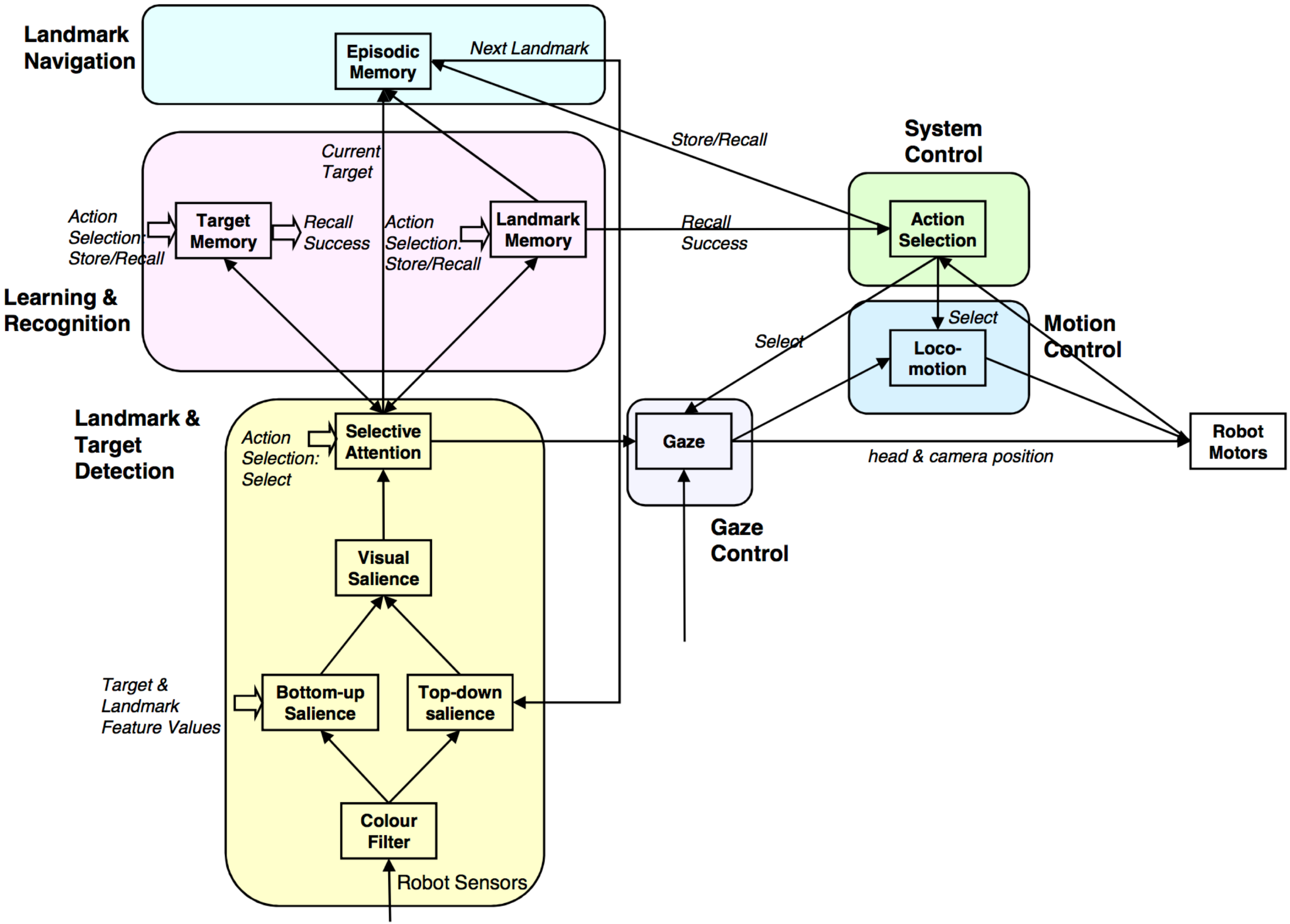
Software Development Life Cycle

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

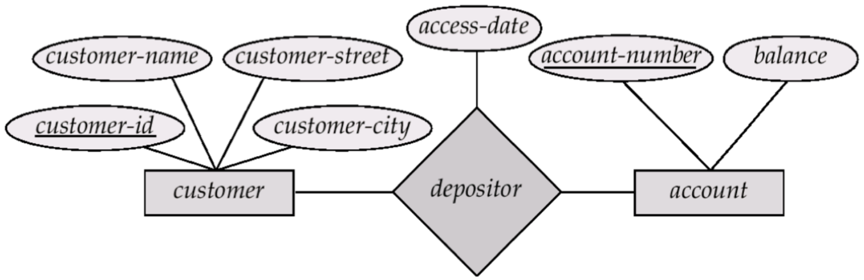
4. System analysis & specification

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 (or context diagram)

Software Development Life Cycle

4. System analysis & specification

   Process-flow model

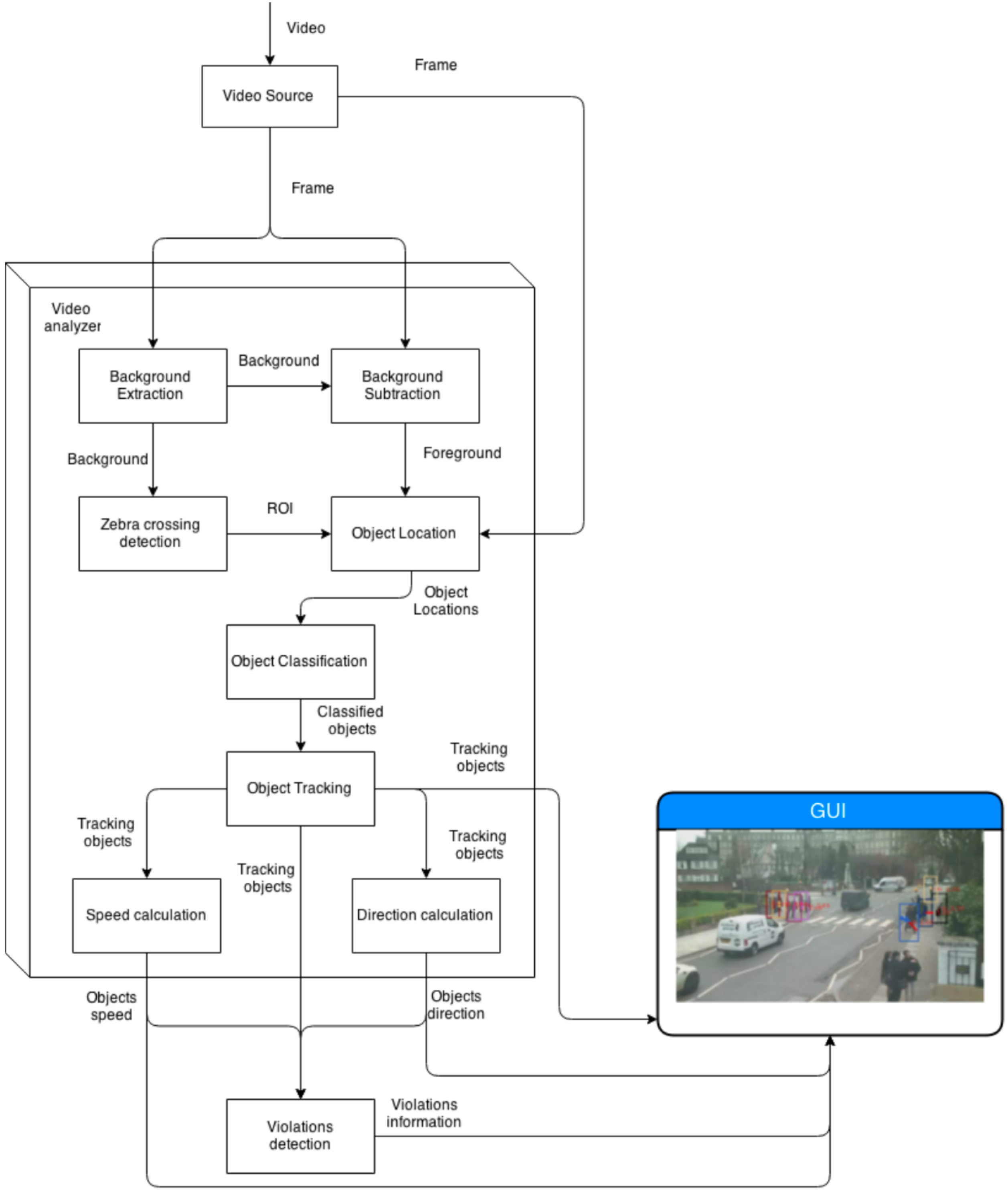
   – DFDs model the transformation of inputs into outputs

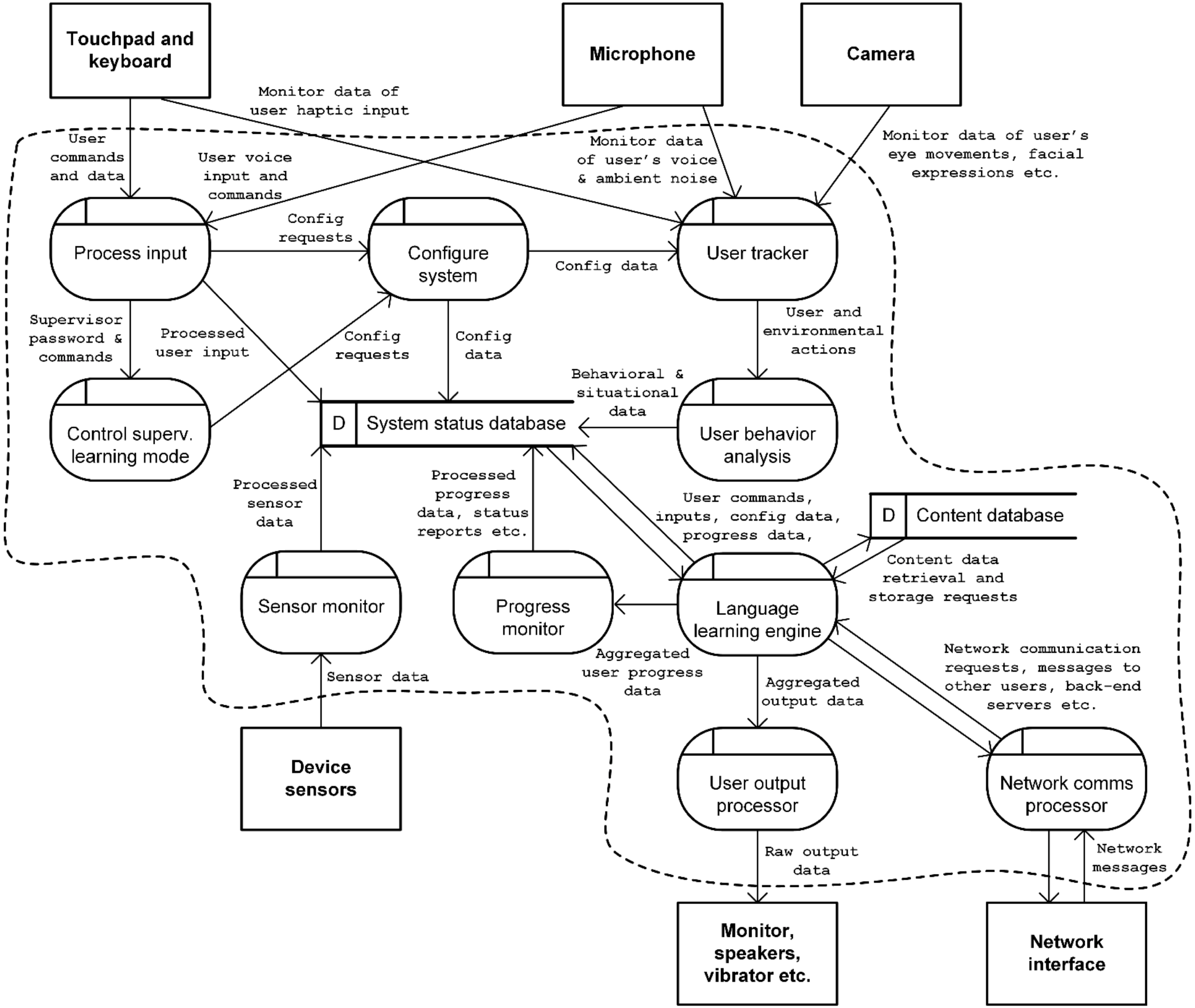
   – Processes/Functions represent individual functions that the system carries out and transform inputs to outputs

   – 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
Tracking objects
Object Tracking
Tracking objects
Tracking objects
Tracking objects
Speed calculation
Direction calculation
GUI
Objects speed
Objects direction
Violations detection
Violations information



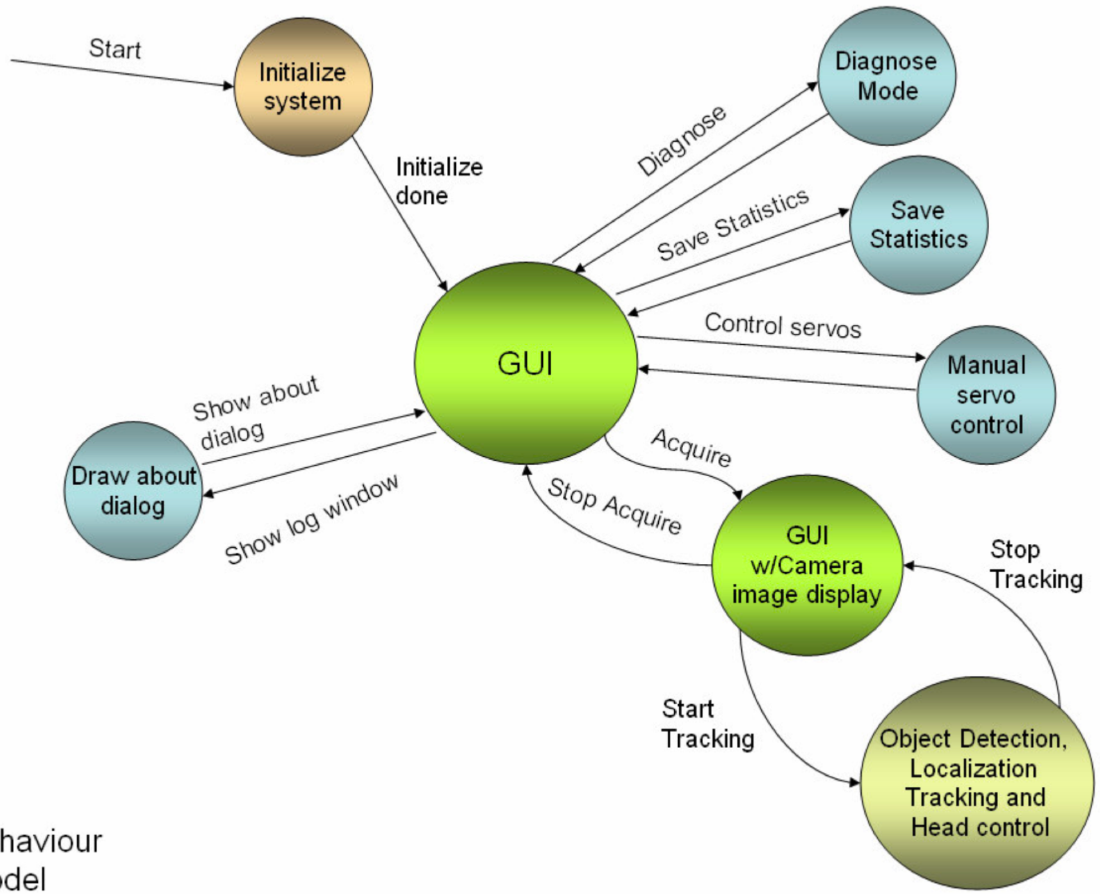
Touchpad and keyboard
Microphone
Camera
Monitor data of user haptic input
User commands and data
User voice input and commands
Monitor data of user's voice & ambient noise
Monitor data of user's eye movements, facial expressions etc.
Process input
Config requests
Configure system
Config data
User tracker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

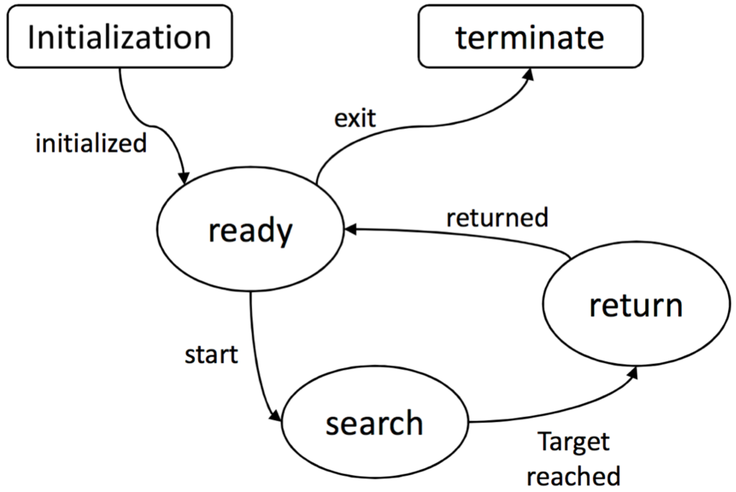
4. System analysis & specification

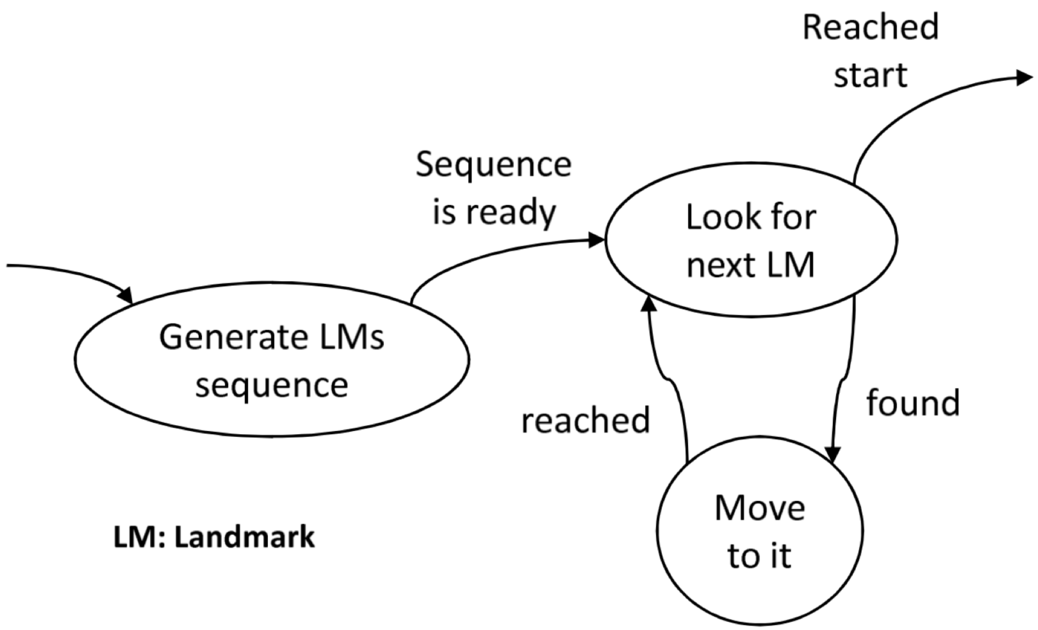
Behavioural model

- 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)



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."

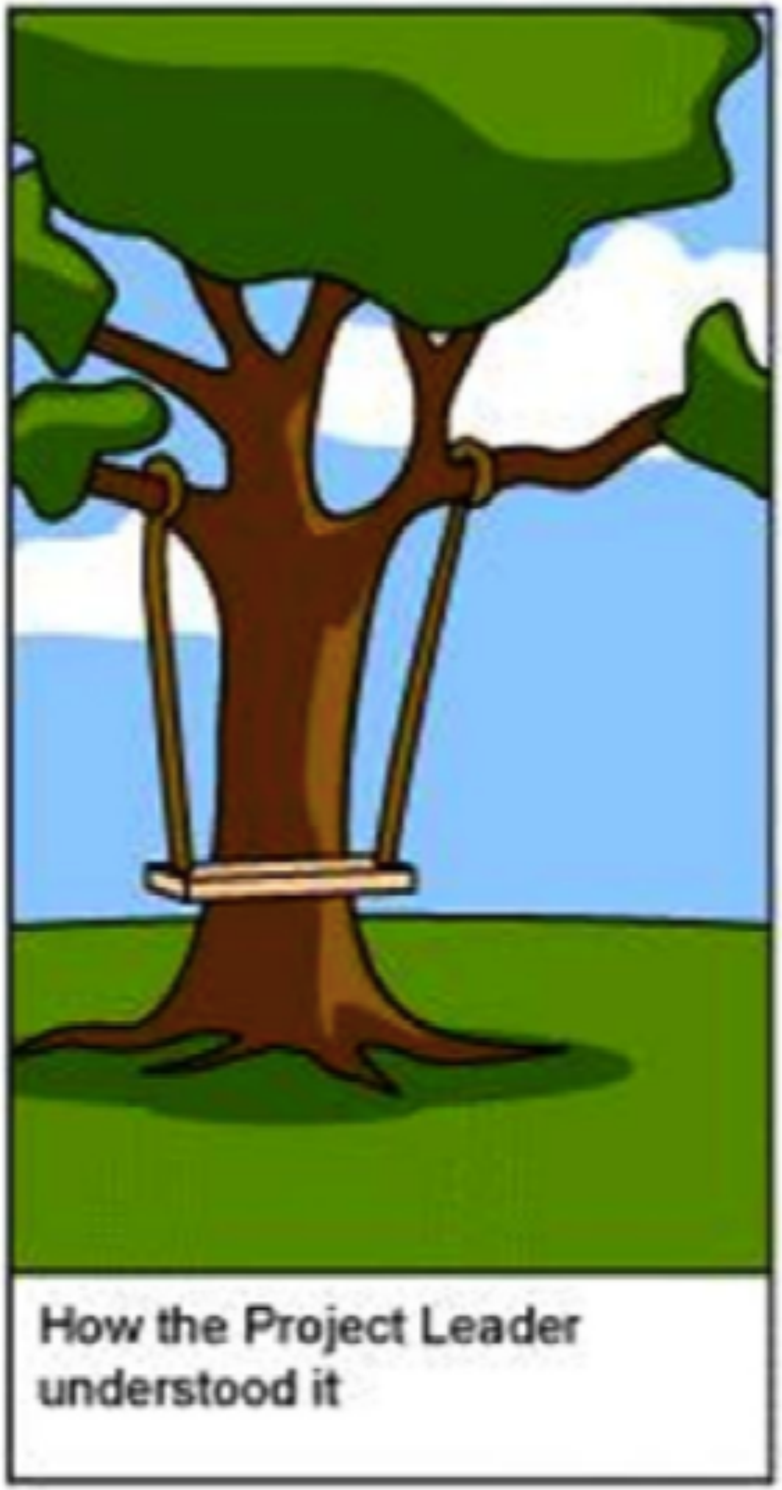
J. A. Whittaker and S. Atkin, "Software Engineering Is Not Enough", IEEE Software, July/August 2002, pp. 108-115.

Software Development Life Cycle

8. Documentation

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

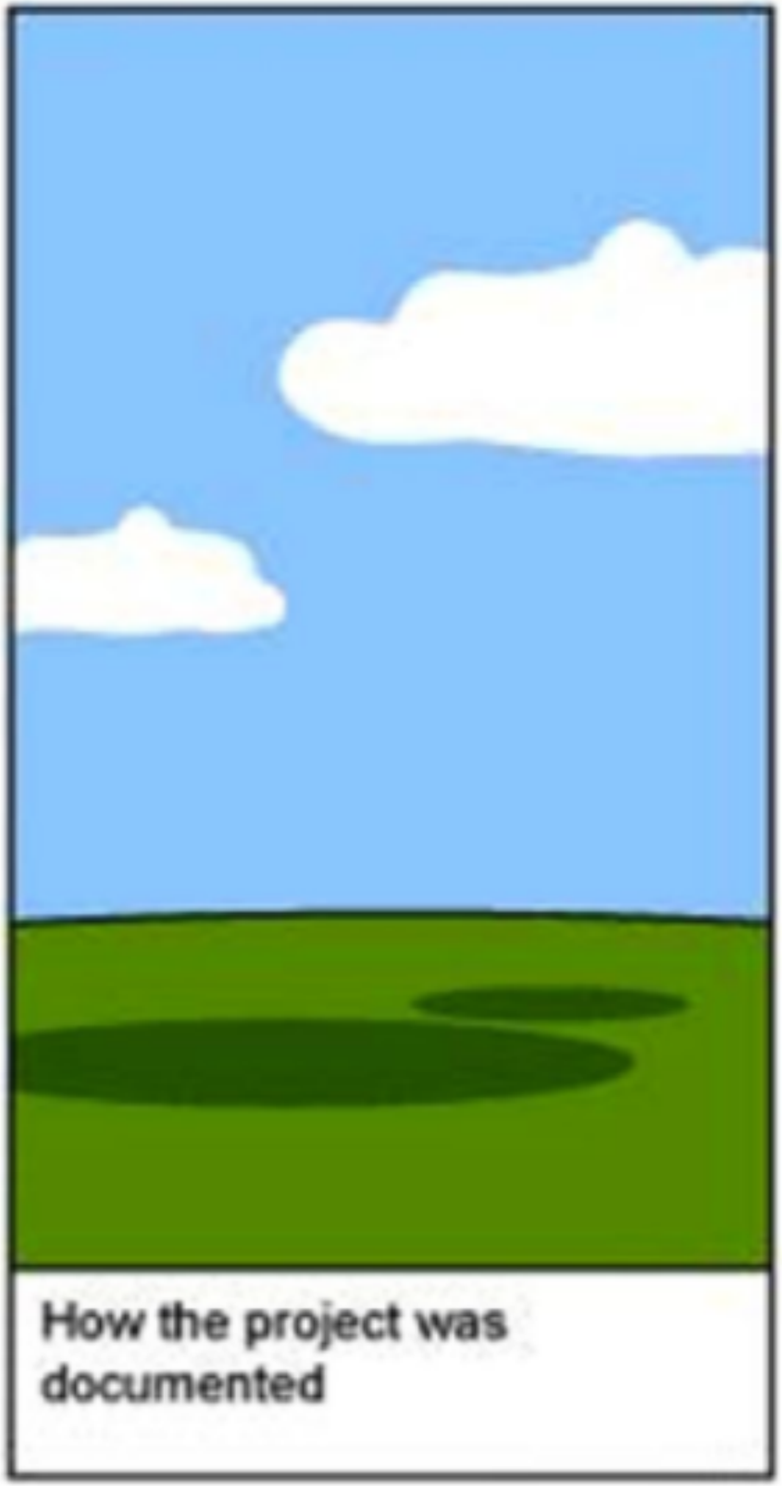

How the customer explained it


How the Project Leader
understood it

How the Analyst designed it

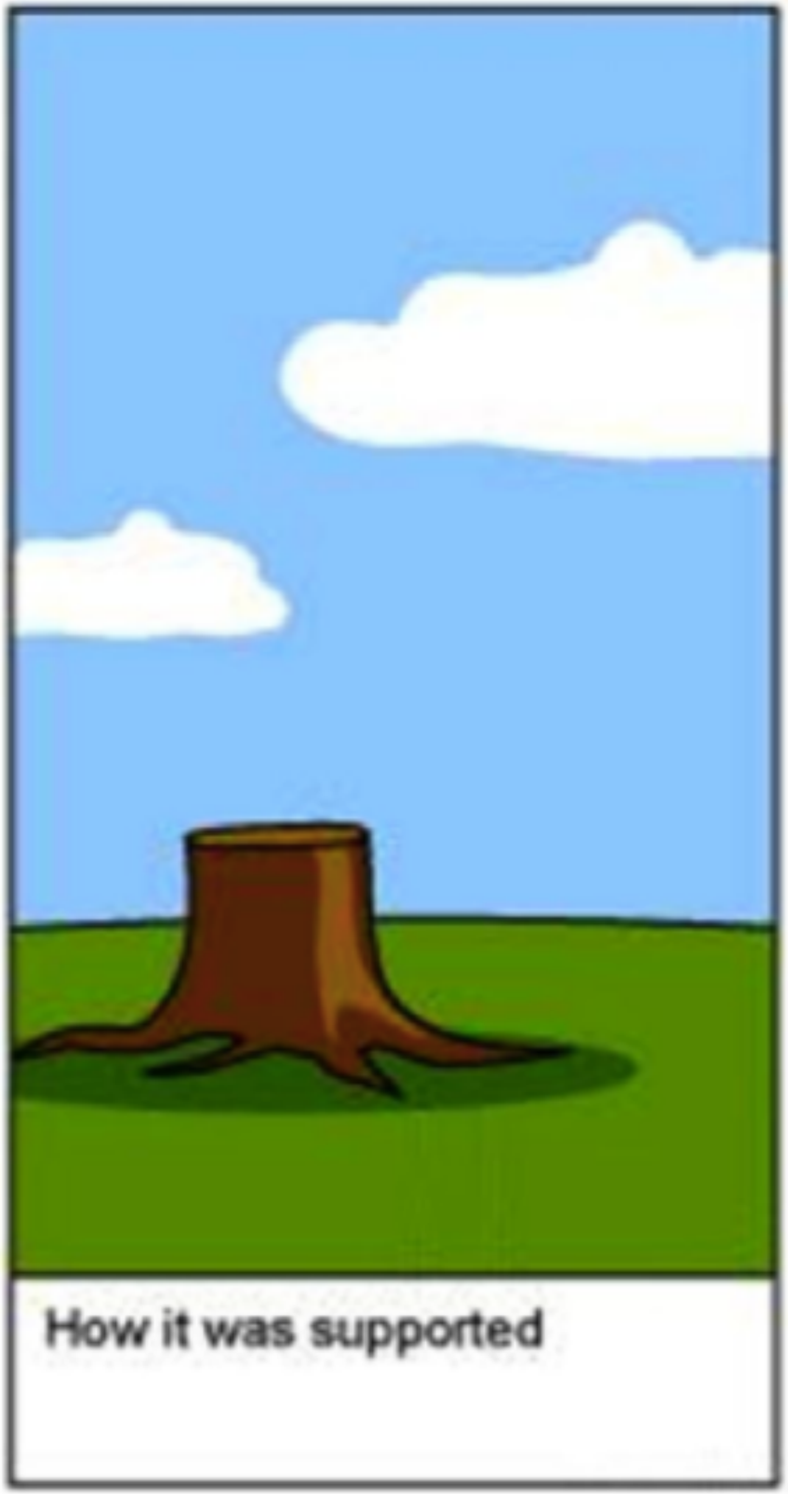
How the Programmer wrote it

How the Business Consultant described it

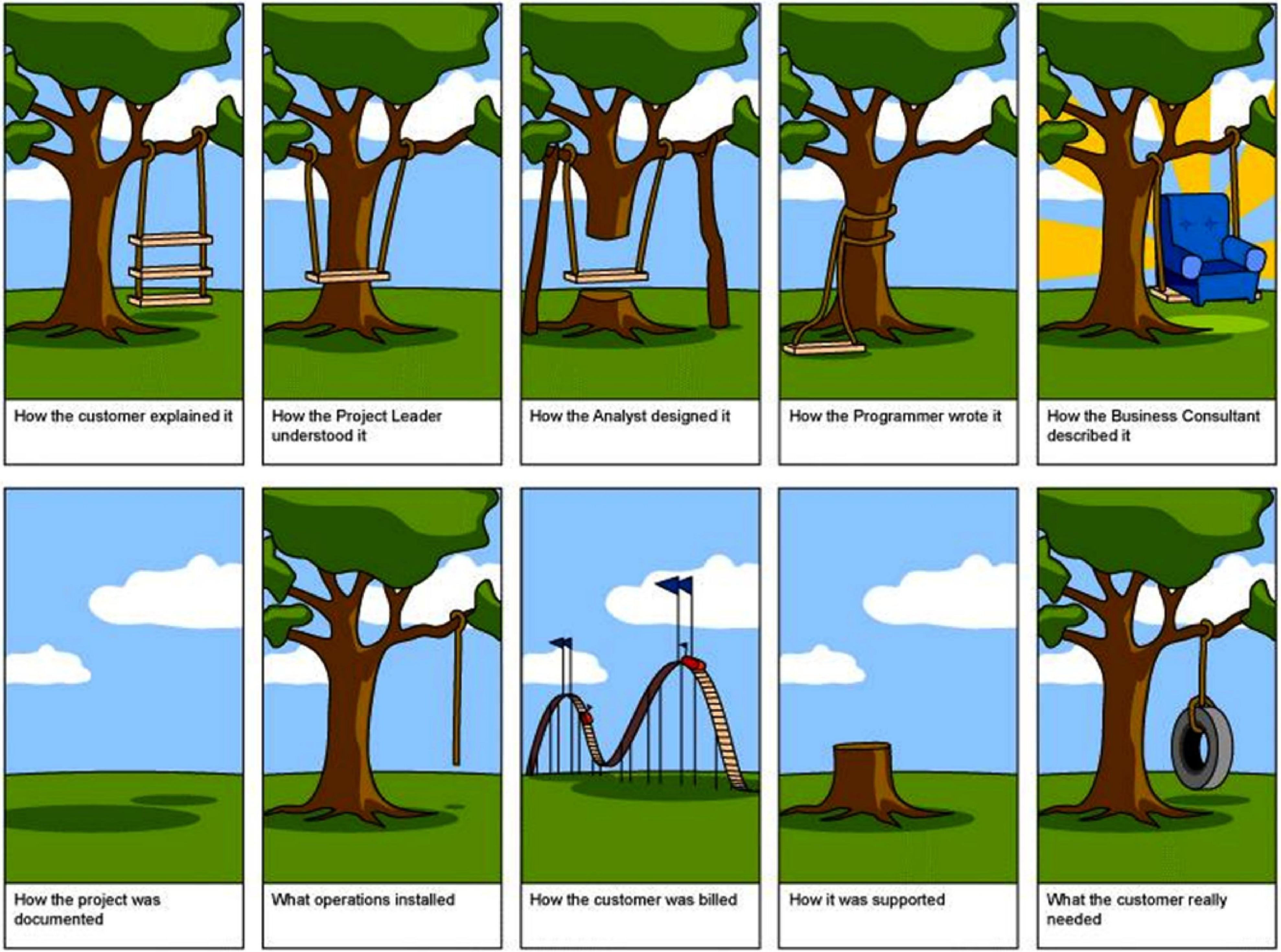

How the project was documented

What operations installed


How the customer was billed


How it was supported

What the customer really needed


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

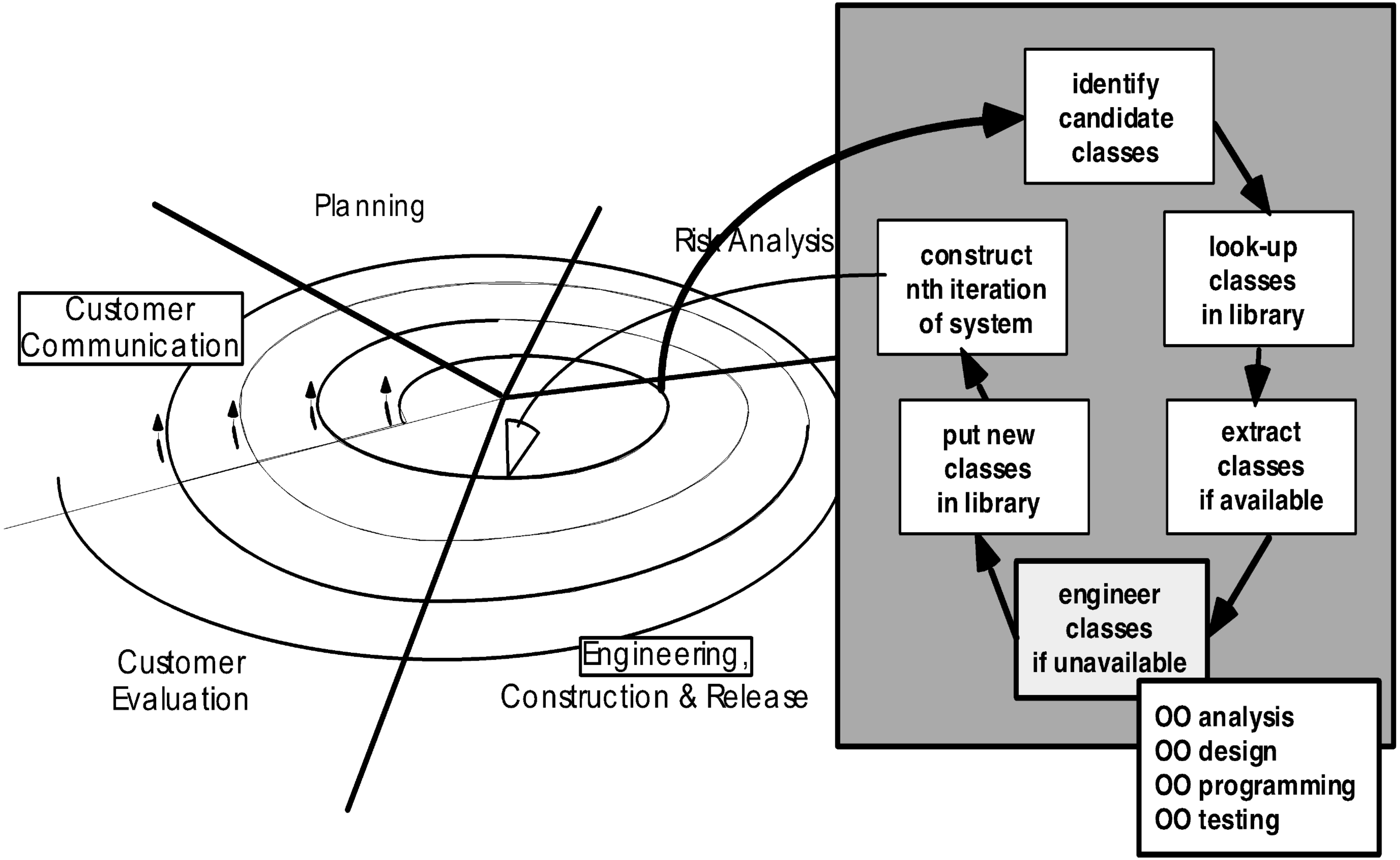
Object-oriented Paradigm

Object-oriented Analysis, Design, and Test

OO classes, inheritance, and polymorphism

- Object technologies include the analysis, design, and testing phases of the development life-cycle, not just OOP
- OO systems tend to evolve over time so an evolutionary process model, such as the spiral model, is probably the best paradigm for OO software engineering

OO classes, inheritance, and polymorphism



OO classes, inheritance, and polymorphism

What is an object-oriented approach?

One definition:

It is the exploitation of class objects, with private data members and associated access functions

OO classes, inheritance, and polymorphism

Key Concept: Class

– A class is a 'template' for the specification of a particular collection of entities (e.g. a widget in a Graphic User Interface).

– More formally, 'a class is an OO concept that encapsulates the data and procedural abstractions that are required to describe the content and behaviour of some real-world entity'.

OO classes, inheritance, and polymorphism

Key Concept: Attributes

- Each class will have specific attributes associated with it (e.g. the position and size of the widget).
- These attributes are queried using associated access functions (e.g. set_position)

OO classes, inheritance, and polymorphism

Key Concept: Object

– An object is a specific instance (or instantiation) of a class (e.g. a button or an input dialogue box).

OO classes, inheritance, and polymorphism

Key Concept: Data Members

- The object will have data members representing the class attributes (e.g. int x, y;)

OO classes, inheritance, and polymorphism

Key Concept: Access function

- The values of these data members are accessed using the access functions (e.g. set_position(x, y);)
- These access functions are called methods (or services).
- Since the methods tend to manipulate a limited number of attributes (i.e. data members) a given class tends to be cohesive.
- Since communication occurs only through methods, a given class tends to be decoupled from other objects.

OO classes, inheritance, and polymorphism

Key Concept: Encapsulation

- The object (and class) encapsulates the data members (attributes), methods (access functions) in one logical entity.

OO classes, inheritance, and polymorphism

Key Concept: Data Hiding

- Furthermore, it allows the implementation of the data members to be hidden (why? Because the only way of getting access to them – of seeing them – is through the methods.) This is called data hiding.

OO classes, inheritance, and polymorphism

Key Concept: Abstraction

- This separation, though data hiding, of physical implementation from logical access is called abstraction

OO classes, inheritance, and polymorphism

Key Concept: Messages

- Objects communicate with each other by sending messages (this just means that a method from one class calls a method from another method and information is passed as arguments).

OO classes, inheritance, and polymorphism

Ellis and Stroustrup define OO as follows:

> ‘The use of derived classes and virtual functions is often called object-oriented programming’

OO classes, inheritance, and polymorphism

Key Concept: Inheritance

- We can define a new class as a sub-class of an existing class
  - e.g. button is a sub-class of the widget class; a toggle button is a sub-class of the button class
- Each sub-class inherits (has by default) the data members and methods of the parent class (the parent class is sometimes called a super-class)
  - For example, both the button and toggle button classes (and objects) have set_position() methods and (private) position data members x and y

OO classes, inheritance, and polymorphism

Key Concept: Inheritance

- A sub-class is sometimes called a derived class
- The C++ programming language allows multiple inheritance, i.e. a sub-class can be derived from two or more super-classes and therefore inherit the attributes and methods from both
  - Multiple inheritance is a somewhat controversial capability as it can cause significant problems for managing the class hierarchy.

OO classes, inheritance, and polymorphism

Key Concept: Polymorphism

- If the super-class is designed appropriately, it is possible to re-define the meaning of some of the super-class methods to suit the particular needs of the derived class

- For example, the widget super-class might have a draw() method. Clearly, we need a different type of drawing for buttons and for input boxes

- However, we would like to use the one generic method draw() for both types of derived classes (i.e. for buttons and for input boxes)

OO classes, inheritance, and polymorphism

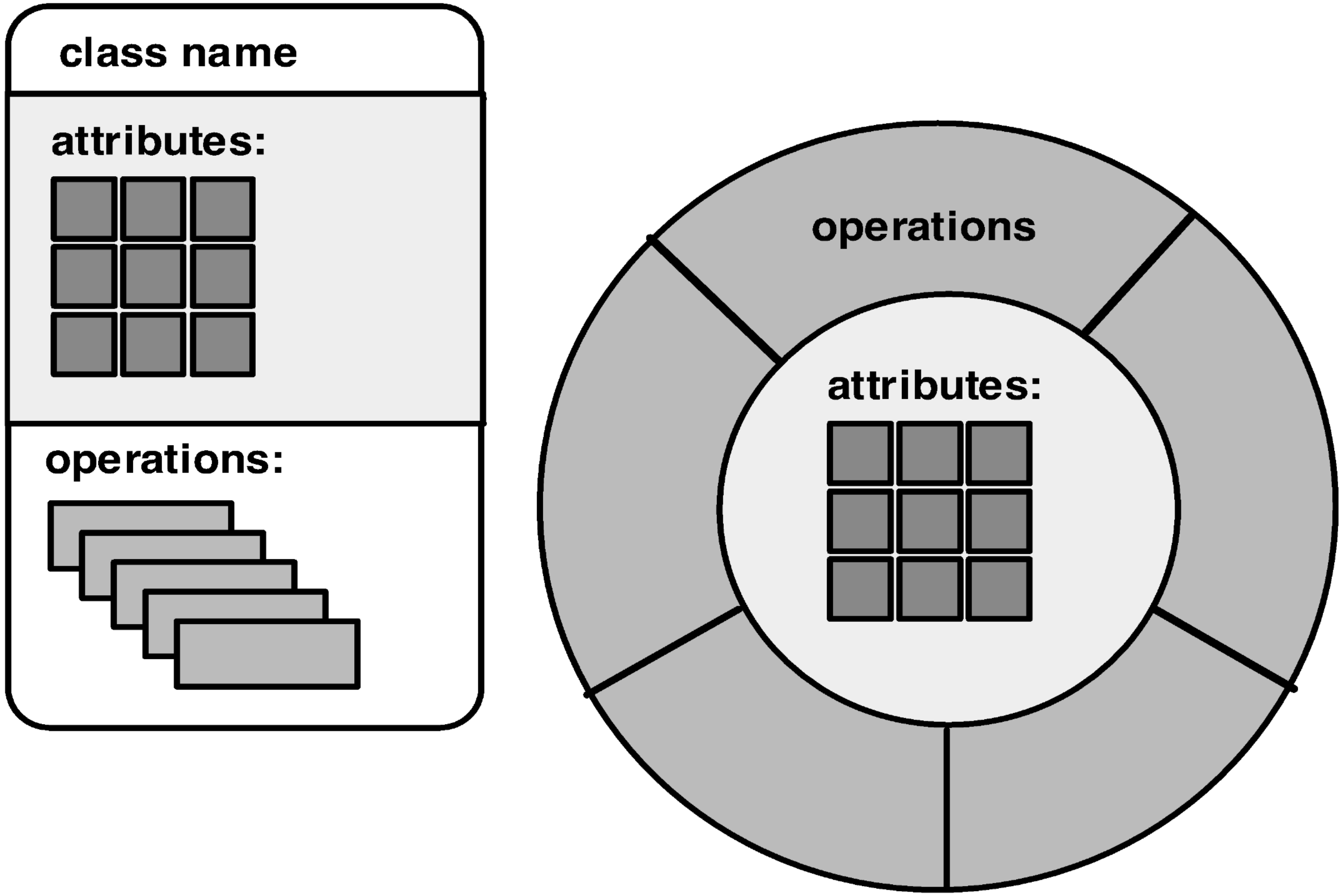
Key Concept: Polymorphism

- OO programming languages allow us to do this. This is called polymorphism (literally: multiple structure) – the ability to define and choose the required method depending on the type of the derived class (or object) without changing the name of the method

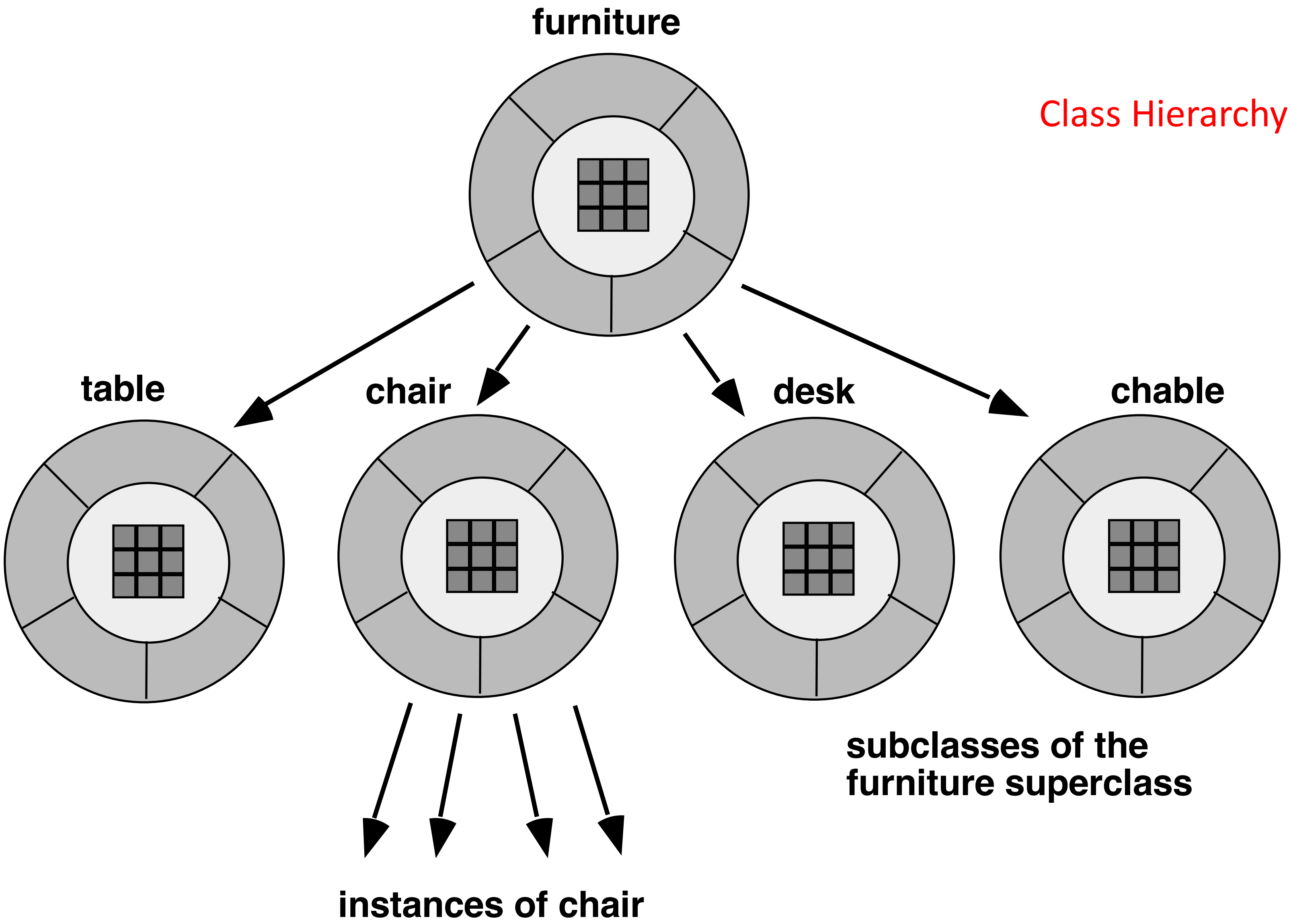
- Thus, the draw() method has many structures, one for each derived class.

OO classes, inheritance, and polymorphism

Two Views of a Class

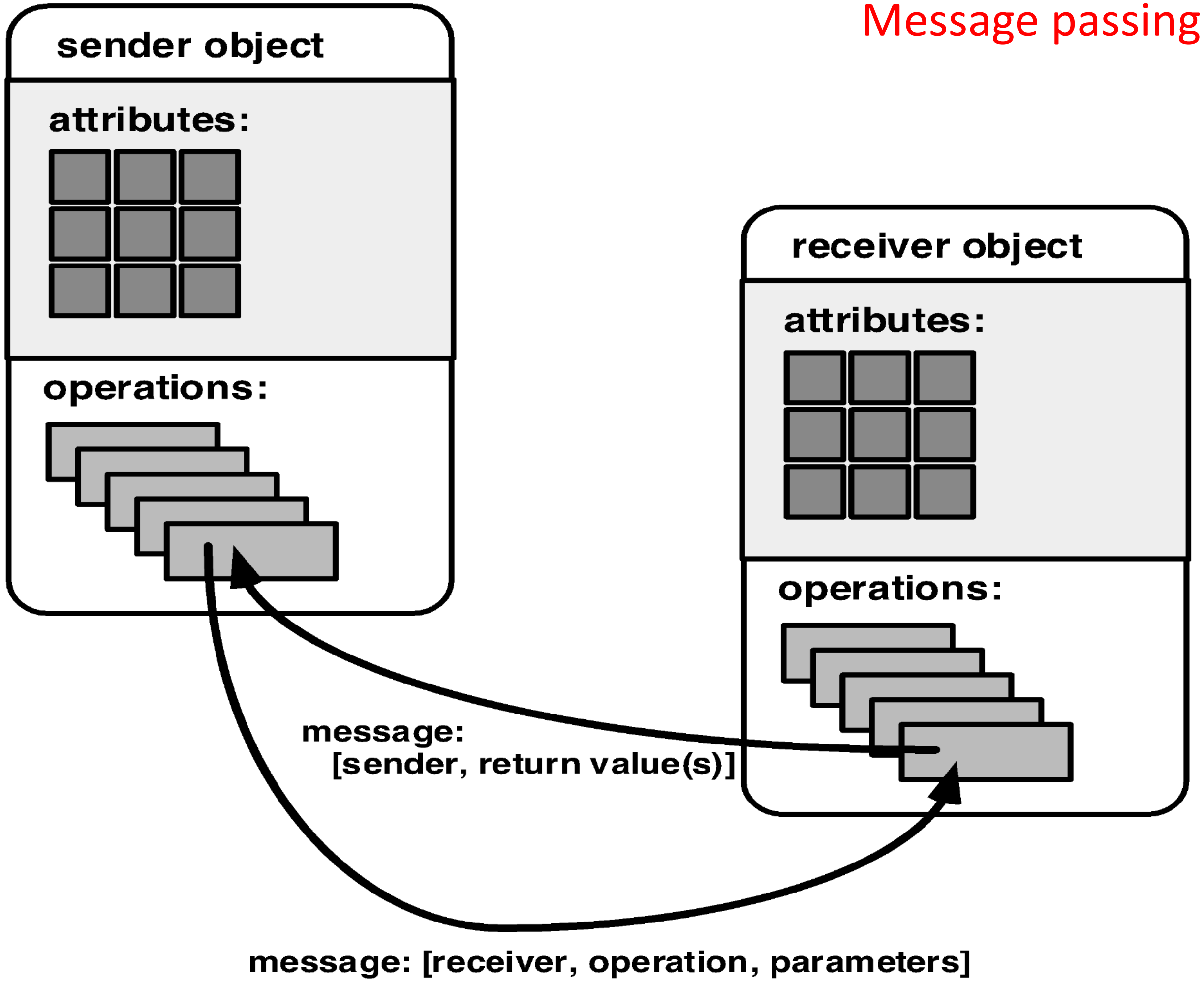


OO classes, inheritance, and polymorphism



OO classes, inheritance, and polymorphism

Message passing between objects



Object-Oriented Analysis - OOA

In order to build an analysis model, five basic principles are applied:

1. The information domain is modeled
2. Module function is described
3. Model behaviour is represented
4. Models are partitioned (decomposed)
5. Early models represent the essence of the problem; later models provide implementation details.

Object-Oriented Analysis - OOA

- The goal of OOA is to define all classes (their relationships and behaviour) that are relevant to the problem to be solved. We do this by:
  - Eliciting user requirements
  - Identifying classes (defining attributes and methods)
  - Specifying class hierarchies
  - Identifying object-to-object relationships
  - Modelling object behaviour

- These steps are reapplied iteratively until the model is complete

Object-Oriented Analysis - OOA

There are many OOA methods. For example:

The Booch Method

- Identify Classes and objects
- Identify the semantics of classes and objects
- Identify relationships among classes and objects
- Conduct a series of refinements
- Implement classes and objects

Object-Oriented Analysis - OOA

The Coad and Yourdon Method

- Identify objects
- Define a generalization-specification structure (gen-spec)
- Define a whole-part structure
- Identify subjects (subsystem components)
- Define attributes
- Define services

Object-Oriented Analysis - OOA

The Jacobson Method (OOSE)

- Relies heavily of use case modeling (how the user (person or device) interacts with the product or system

Object-Oriented Analysis - OOA

The Rambaugh Method (Object Modelling Technique OMT)

- Scope the problem
- Build an object model
- Develop a dynamic model
- Construct a functional model

Object-Oriented Analysis - OOA

There are seven generic steps in OOA:

1. Obtain customer requirements

Identify scenarios or use cases;
build a requirements model

2. Select classes and objects using basic requirements
3. Identify attributes and operations for each object
4. Define structures and hierarchies that organize classes
5. Build an object-relationship model
6. Build an object-behaviour model
7. Review the OO analysis model against use cases / scenarios

Object-Oriented Analysis - OOA

Requirements Gathering and Use Cases

- Use cases are a set of scenarios each of which identifies a thread of usage of the system to be constructed

- They can be constructed by first identifying the actors: the people or devices that use the system (anything that communicates with the system)

- Note that an actor is not equivalent to a user: an actor reflects a particular role (a user may have many different roles, e.g. in configuration, normal, test, maintenance modes)

Object-Oriented Analysis - OOA

Requirements Gathering and Use Cases

– Once the actors have been identified, on can then develop the use case, typically by answer the following questions:

1. What are the main tasks or functions that are performed by the actor?
2. What system information will the actor require, produce, or change?
3. Will the actor have to inform the system about changes in the external environment?
4. What information does the actor desire from the system?
5. Does the actor wish to be informed about unexpected changes?

Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

- CRC modeling provides a simple means for identifying and organizing the classes that are relevant to a system
- Responsibilities are the attributes and operations that are relevant for the class ('a responsibility is anything a class knows or does')
- Collaborators are those classes required to provide a class with the information needed to complete a responsibility (a collaboration implies either a request for information or a request for some action)

Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

- Guidelines for Identifying Classes
  - We said earlier that 'a class is an OO concept that encapsulates the data and procedural abstractions that are required to describe the content and behaviour of some real-world entity'.
  - We can classify different types of entity and this will help identify the associated classes:
  - Device classes:
    these model external entities such as sensors, motors, and key-boards.
  - Property classes:
    these represent some important property of the problem environment (e.g. credit rating)
  - Interaction classes:
    these model interactions what occur among other objects (e.g. purchase of a commodity).

Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

- Guidelines for Identifying Classes
  - In addition, objects/classes can be categorized by a set of characteristics:
  - Tangibility
    does the class respresent a real device/physical object or does it represent abstract information?
  - Inclusiveness
    is the class atomic (it does not include other classes) or is it an aggregate (it includes at least one nested object)?
  - Sequentiality
    is the class concurrent (i.e. it has its own thread of control) or sequential (it is controlled by outside resources)?

Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

– Guidelines for Identifying Classes

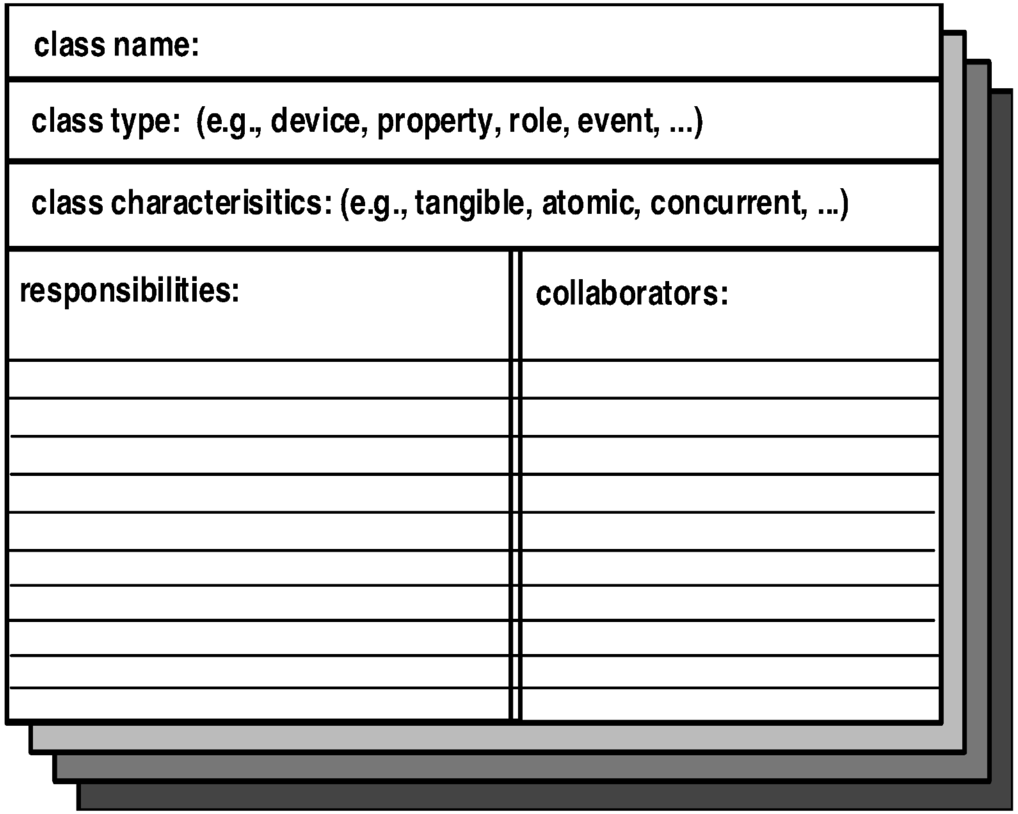
- Persistence
  is the class *transient* (i.e. is it created and removed during program operation); *temporary* (it is created during program operation and removed once the program terminates) or *permanent* (it is stored in, e.g., a database)?

- Integrity
  is the class corruptible (i.e. it does not protect its resources from outside influence) or it is guarded (i.e. the class enforces controls on access to its resources)?

- For each class, we complete a CRC 'index card' noting these class types and characteristics, and listing all the collaborators and attributes for the class.

Object-Oriented Analysis - OOA

- Class-Responsibility-Collaborator (CRC) Modelling



Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

– Guidelines for assigning responsibilities to classes

- System intelligence should be evenly distributed.
- Each responsibility should be stated as generally as possible.
- Information and the behavior that is related to it should reside within the same class.
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.

Object-Oriented Analysis - OOA

Class-Responsibility-Collaborator (CRC) Modelling

- Reviewing the CRC Model

  - All participants in the review (of the CRC model) are given a subset of the CRC model index cards.
  - All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
  - The review leader reads the use-case deliberately. As the review leader comes to a named object, she passes the token to the person holding the corresponding class index card.
  - When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
  - If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards

Object-Oriented Analysis - OOA

Defining Structures and Hierarchies

- The next step is to organize the classes identified in the CRC phase into hierarchies

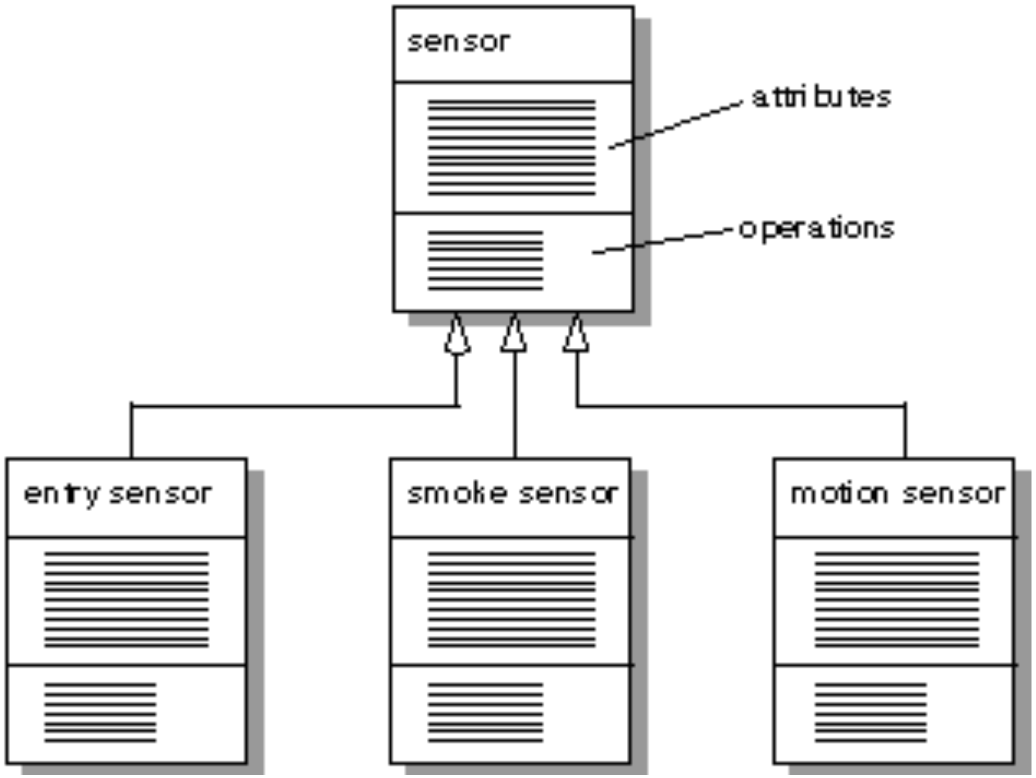
- There are two types of hierarchy:

  1. Generalization-Specialization (Gen-Spec) structure

  2. Composite-Aggregate (Whole-Part) structure

Object-Oriented Analysis - OOA

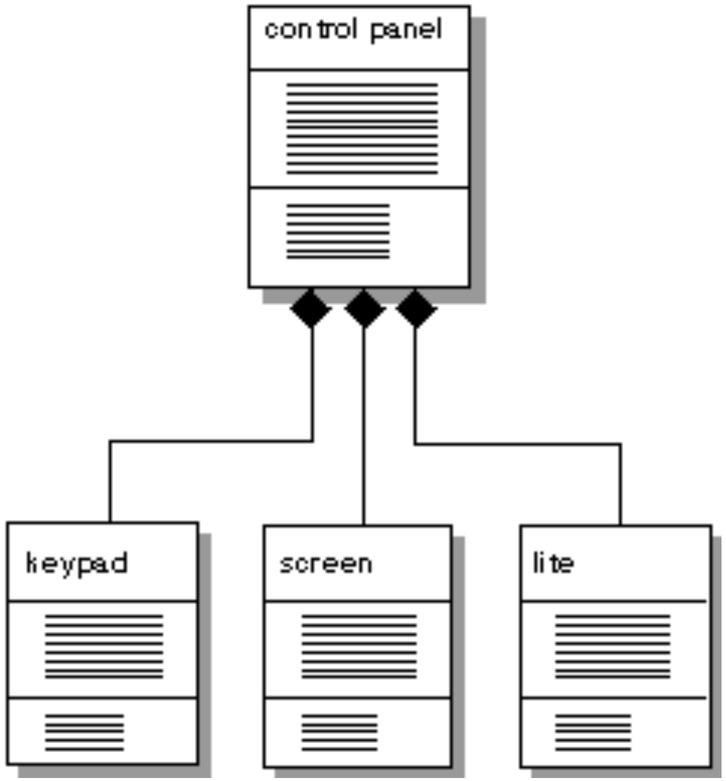
Gen-Spec Hierarchy



The relationship between classes in a Gen-Spec hierarchy can be viewed as a 'Is A' relation

Object-Oriented Analysis - OOA

Composite-Aggregate Hierarchy



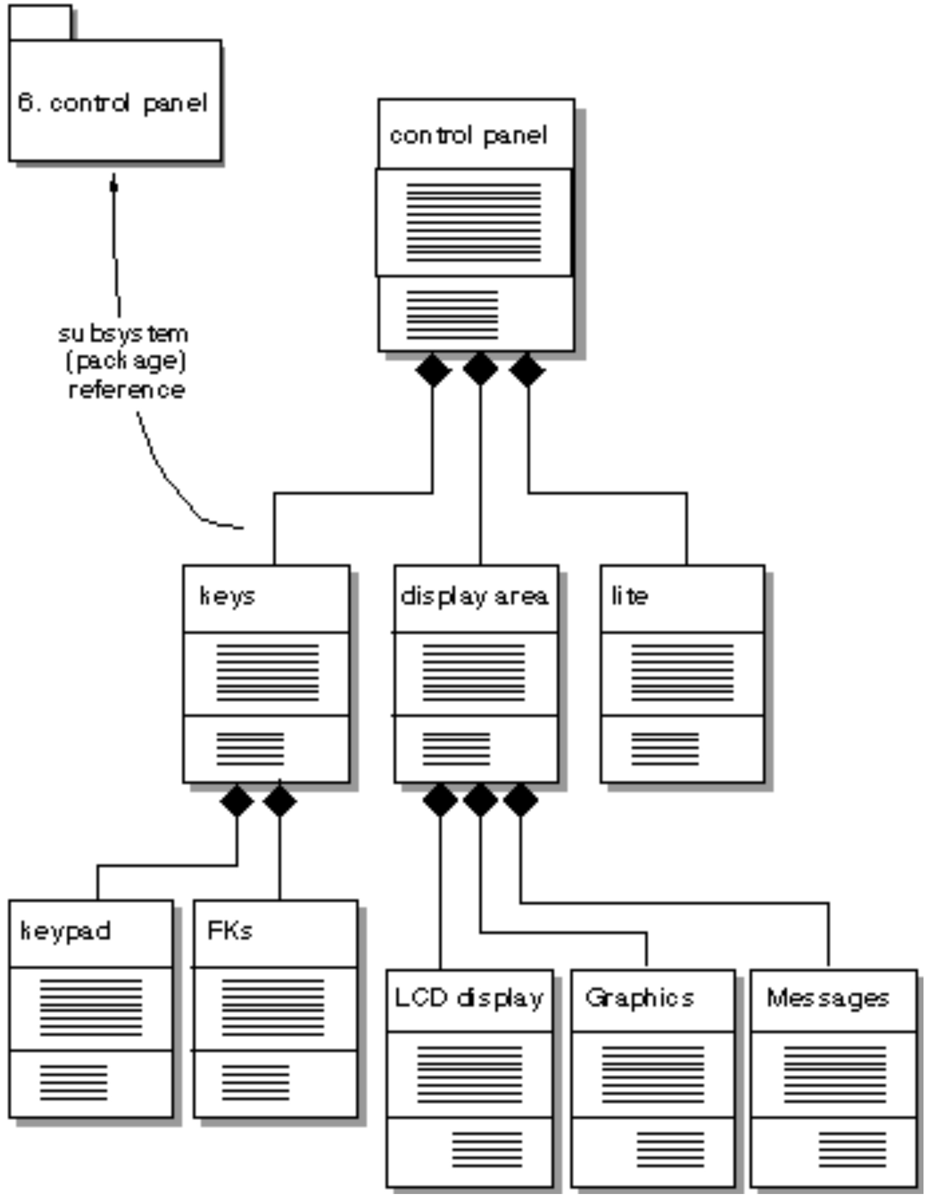
The relationship between classes in a composite-aggregate hierarchy can be viewed as a ‘Has A’ relation

Object-Oriented Analysis - OOA

Defining Subjects and Subsystems

- Once the class hierarchies have been identified, we then try to group them into subsystems or subjects
- A subject / subsystem is a subset of classes that collaborate among themselves to accomplish a set of cohesive responsibilities
- A subsystem / subject implements one or more contracts with its outside collaborators
- A contract is a specific list of requests that collaborators can make of the subsystems

Object-Oriented Analysis - OOA



Object-Oriented Analysis - OOA

The Object-Relationship Model

– The next step is to define those collaborator classes that help in achieving each responsibility.

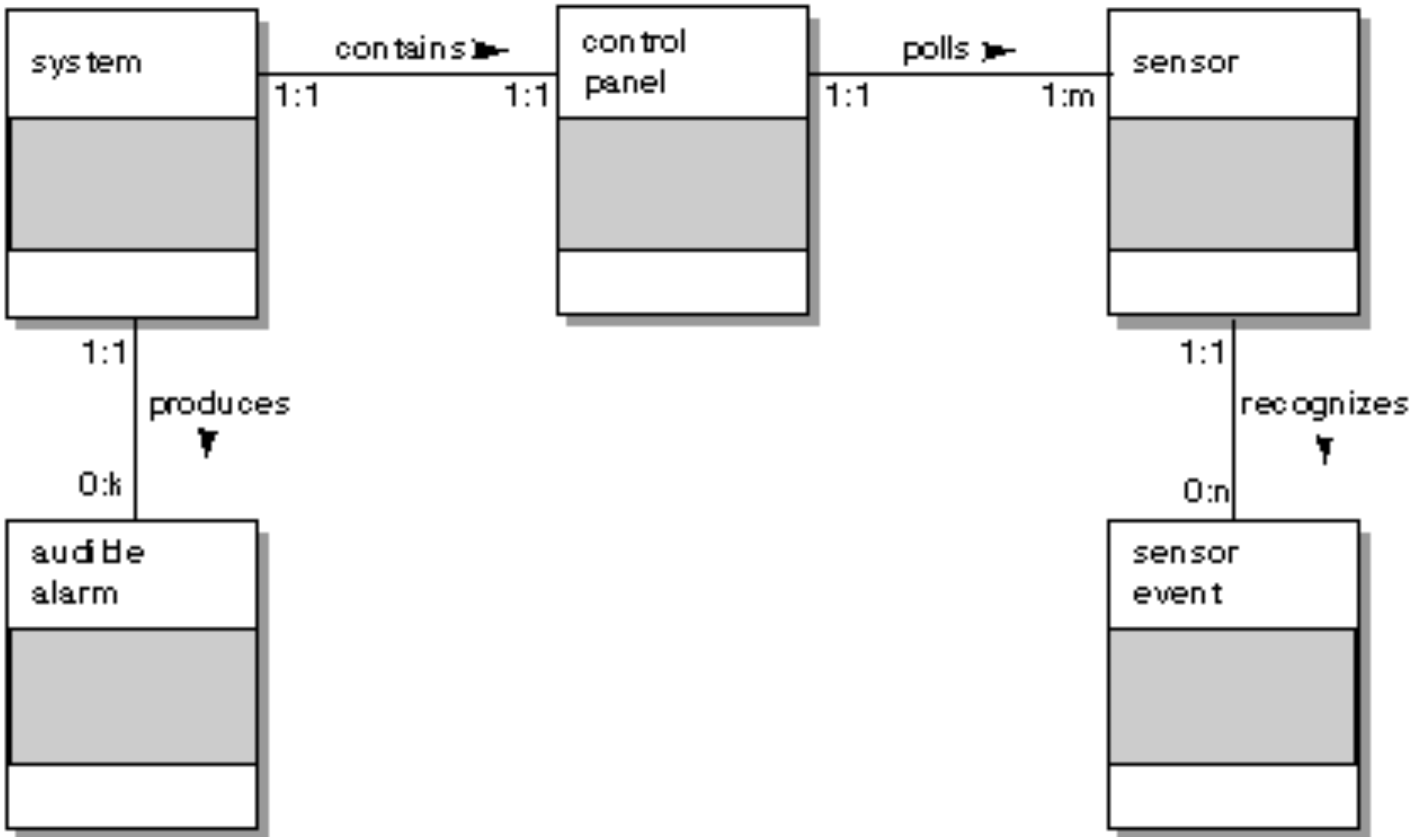
– This establishes a connection between classes. A relationship exists between any two classes that are connected

– There are many different (but equivalent) graphical notations for representing the object-relationship model. All are the same as the entity-relationship diagrams that are used in modeling database systems and they depict the existence of a relationship (line) and the cardinality of the relationship (1:1, 1:n, n:n, etc).

– In the following notation, the direction of the relation is also shown and the cardinality is show at both ends of the relationship line. A cardinality of zero implies a partial participation.

Object-Oriented Analysis - OOA

The Object-Relationship Model



Object-Oriented Analysis - OOA

The Object-Behaviour Model

– The object-behaviour model indicates how an OO system will respond to external events or stimuli

– To create the model, the you should perform the following steps:

1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create an event trace for each use-case.
4. Build a state transition diagram for the system.
5. Review the object-behavior model to verify accuracy and consistency.

Object-Oriented Analysis - OOA

The Object-Behaviour Model

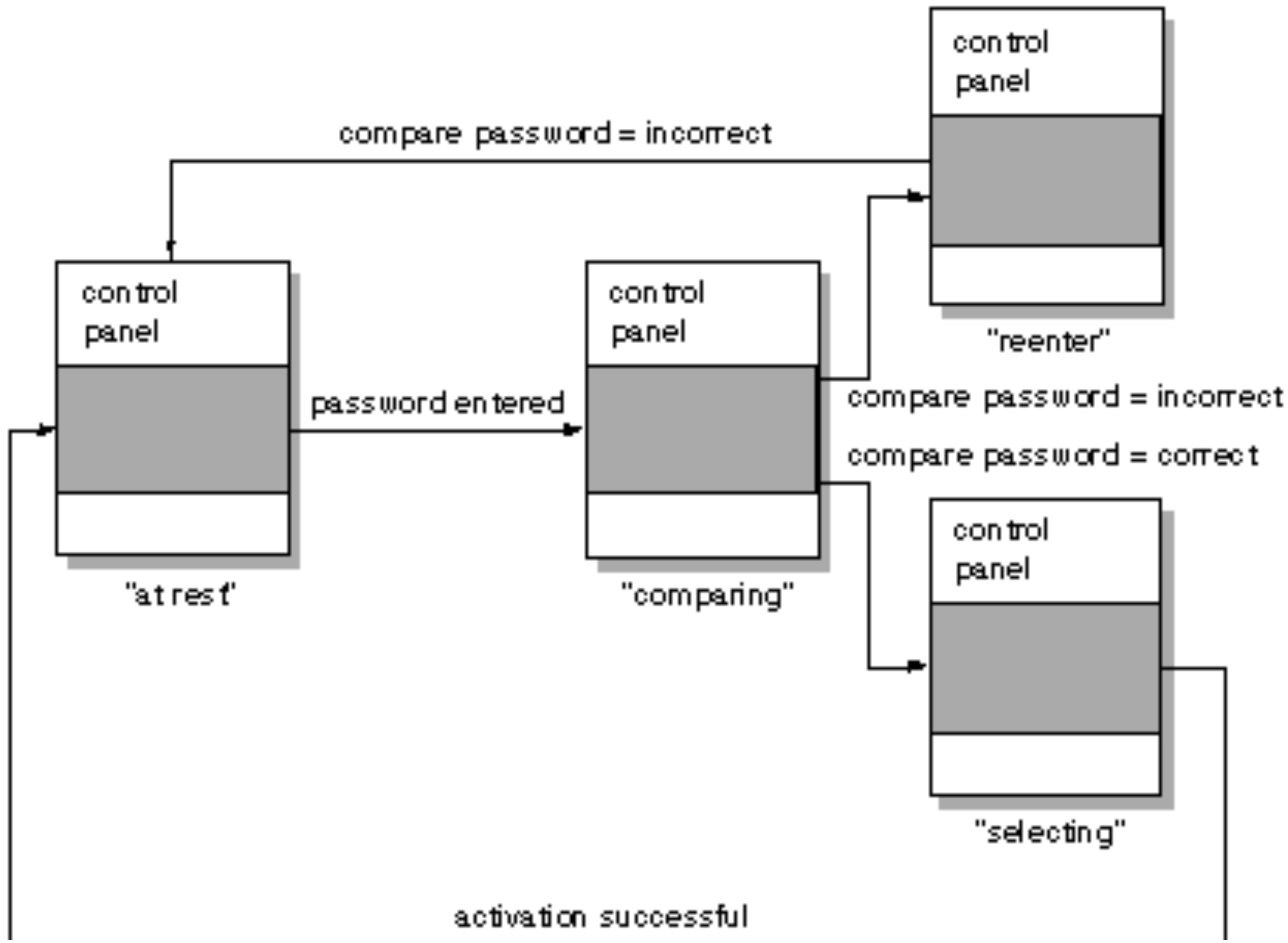
- In general, an event occurs whenever an OO system and an actor exchange information.
- Note that an event is Boolean: an event is not the information that has been exchanged; it is the fact that information has been exchanged.
- An actor should be identified for each event; the information that is exchanged should be noted and any conditions or constraints should be indicated.
- Some events have an explicit impact on the flow of control of the use case, other have no direct impact on the flow of control.

Object-Oriented Analysis - OOA

The Object-Behaviour Model

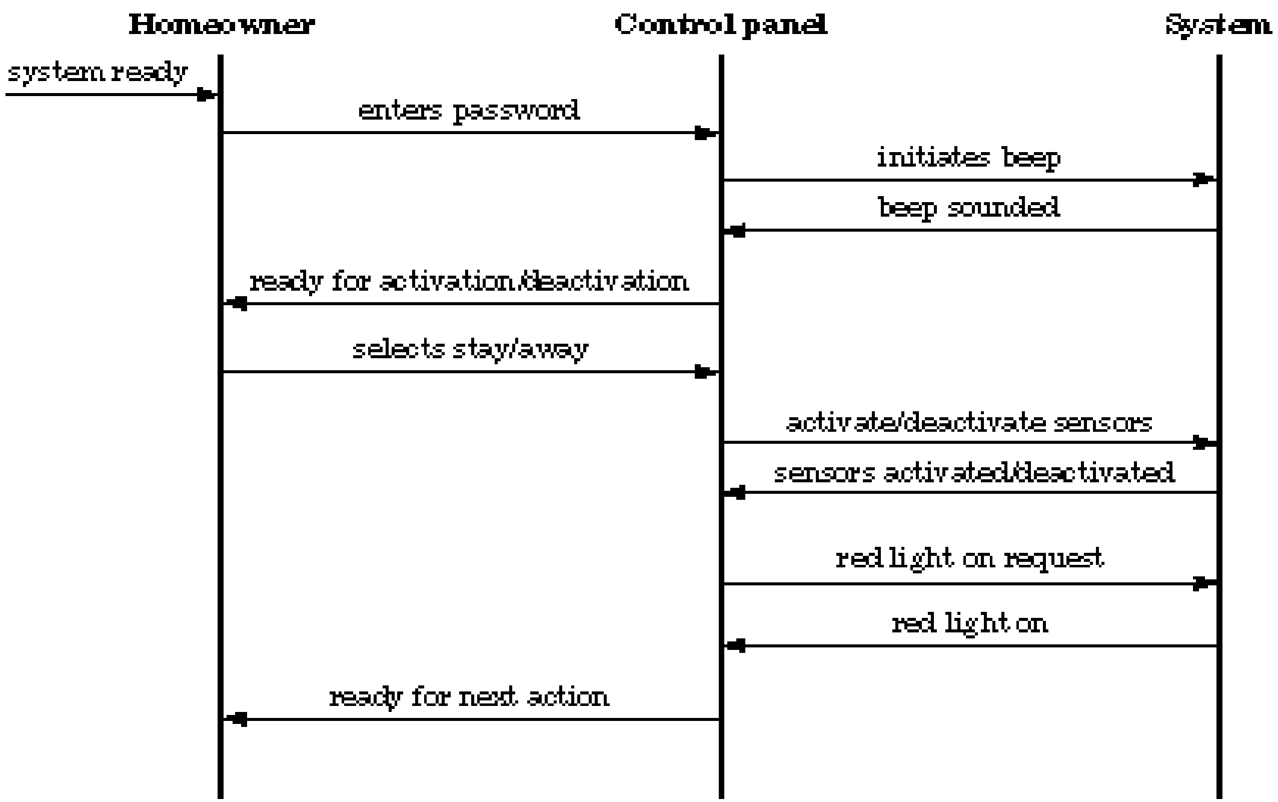
- For OO systems, two different characterizations of state must be considered
    1. The state of each object as the system performs its function.
    2. The state of the system as observed from outside as the system performs its function
- The state of an object can be both passive and active:
    - The passive state is the current status of all an object's attributes.
    - The active state is the current status of the object as it undergoes a continuing transformation or process.
- An event (i.e. a trigger) must occur to force an object to make a transition from one active state to another.

Object-Oriented Analysis - OOA



A partial active state transition diagram for the object *control panel*

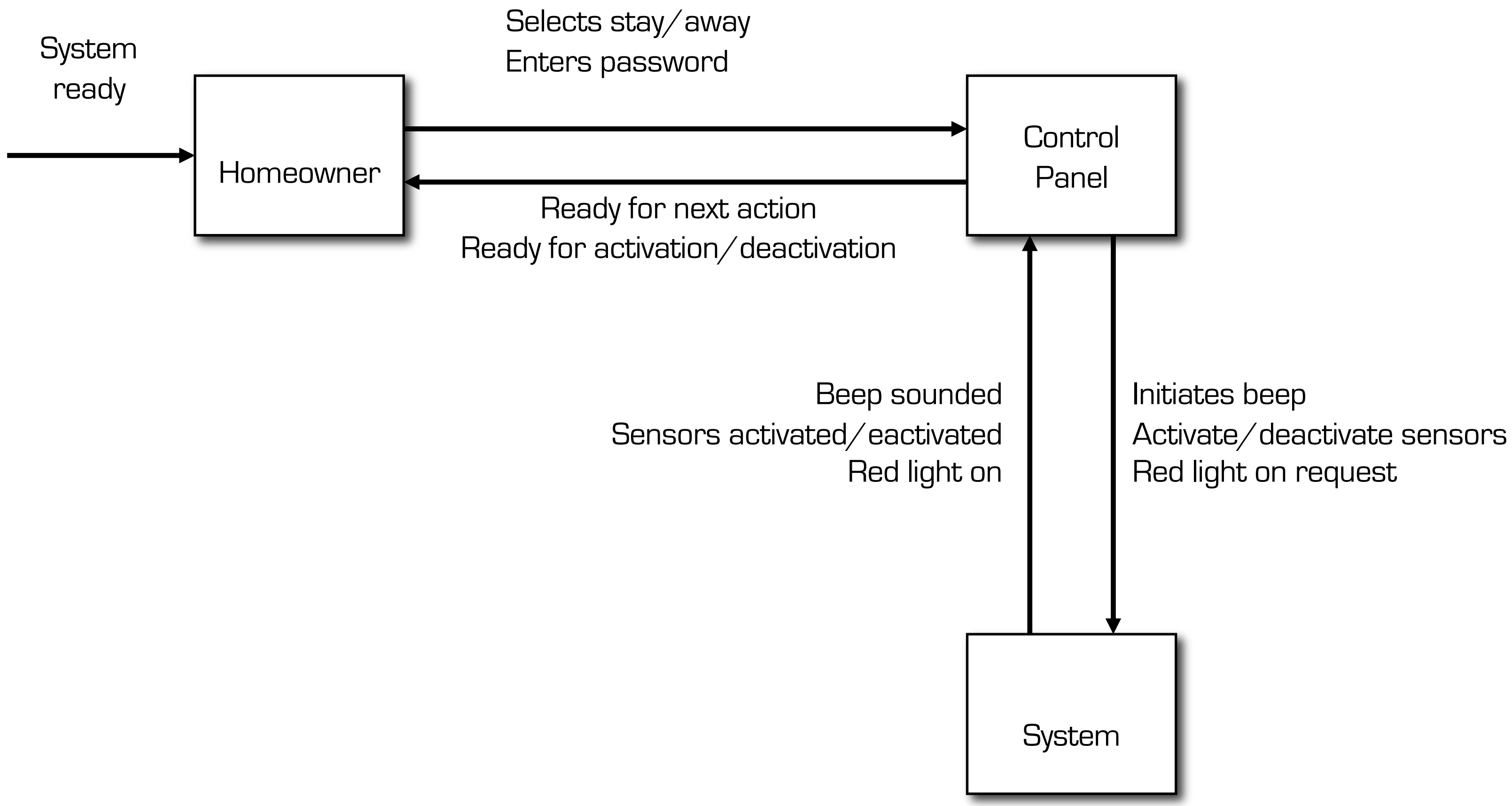
Object-Oriented Analysis - OOA



An Event Trace model: indicates how events cause transitions from object to object

An event trace is actually a shorthand version of the use case

Object-Oriented Analysis - OOA



Event flow diagram: summary of all of the events that cause transitions between objects

Object-Oriented Design - OOD

'Designing object-oriented software is hard, and designing reusable object-oriented software is even harder … a reusable and flexible design is difficult if not impossible to get "right" the first time'

E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software, 1994*

OOD is a part of an iterative cycle of analysis and design

Several iterations of which may be required before one proceeds to the OOP stage.

Object-Oriented Design - OOD

- There are many OOD approaches, almost all of which are the direct adjuncts of OOA approaches (e.g. the Booch method, the Coad and Yourdon Method, the Jacobson method, the Rambaugh method)

- The following gives just an overview of the issues that are common to all approaches

- OOD is a critical process in the transition from OOA model to OO implementation (OO programming) because it requires you to set out the details of all aspects of the OOA model that will be needed when you come to write the code. At the same time, it allows you to validate the OOA model

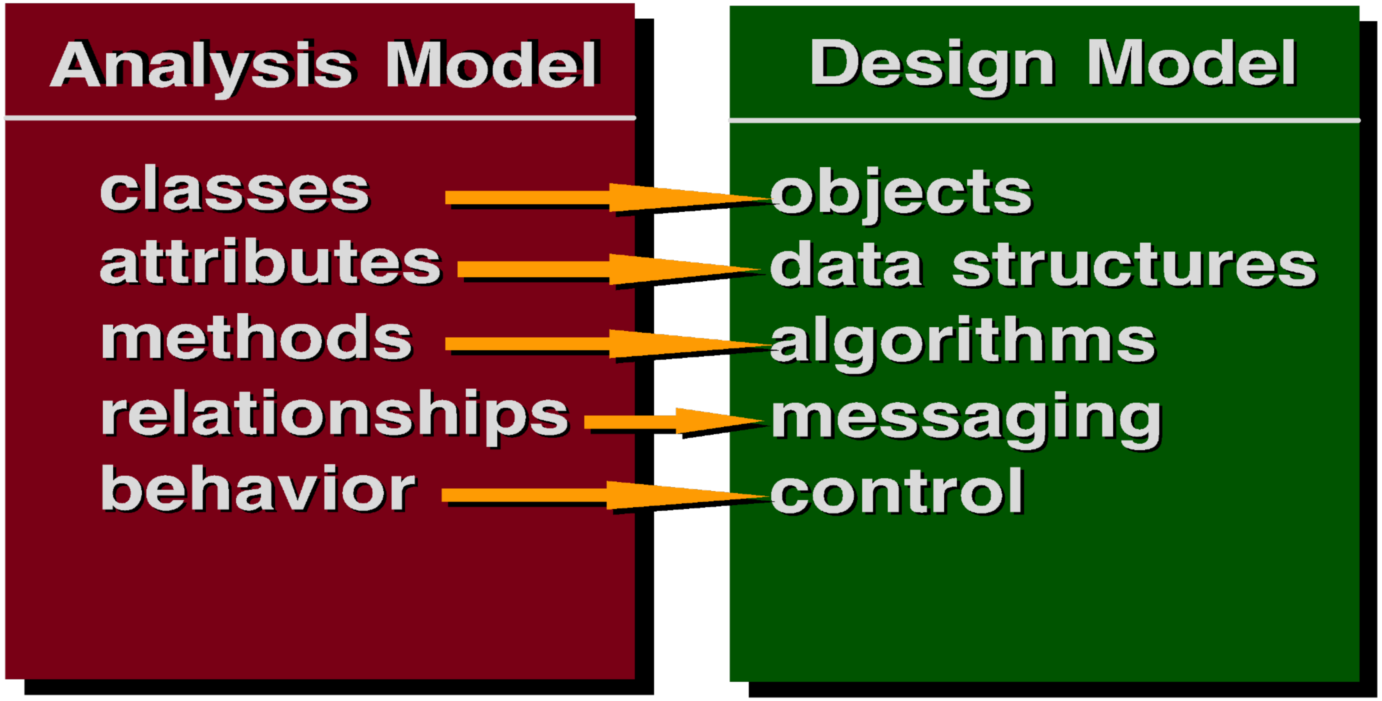
Object-Oriented Design - OOD

- The main goal in OOD is to make the OOA models less abstract by filling in all the details, but without going as far as writing the OO code

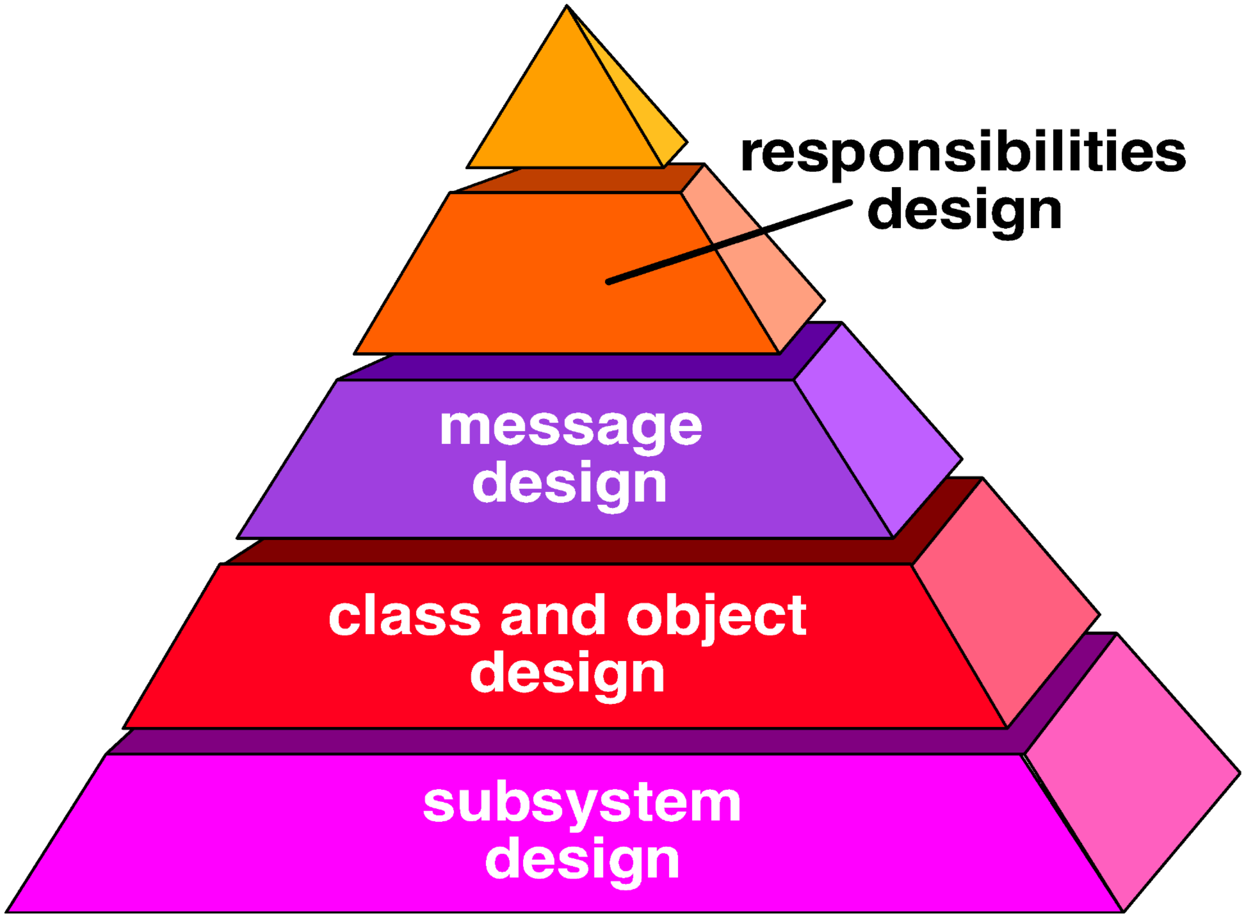
- This will require you to state exactly:
  - how the attributes (data members) will be represented;
  - the algorithms and calling sequences required to effect the methods;
  - the protocols for effecting the messaging between the objects;
  - the control mechanism by which the system behaviour will be achieved (i.e. task management and HCI management)

- We focus on the algorithmic, representation, and interface issues; on how the system will be implemented, rather than on what will be implemented.

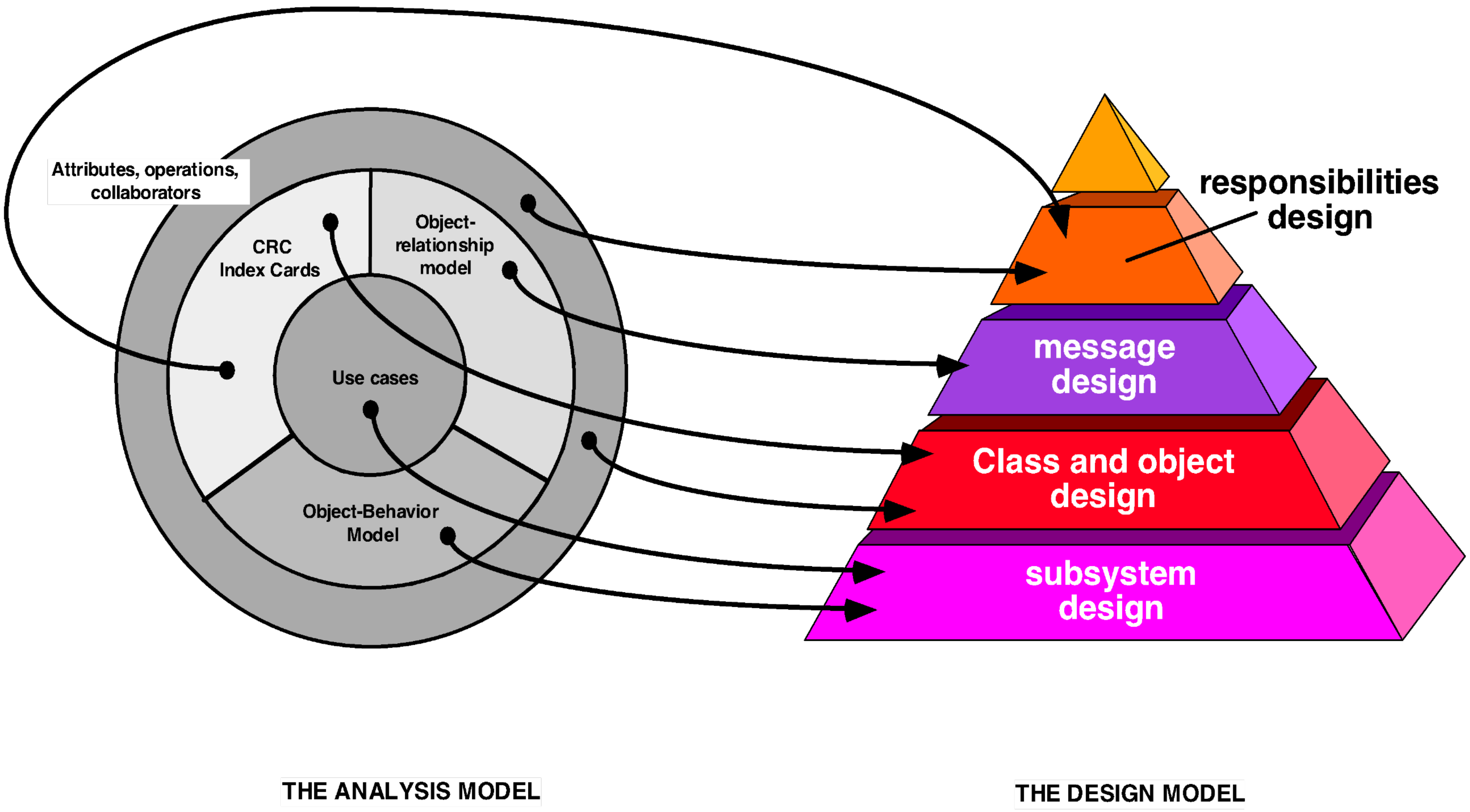
Object-Oriented Design - OOD



Object-Oriented Design - OOD



Object-Oriented Design - OOD



Object-Oriented Design - OOD

The software engineering will follow some standard steps in the design process:

– Partition the analysis model into sub-systems
– Identify concurrency that is dictated by the problem
– Allocate subsystems to processors and tasks
– Choose a basic strategy for implementing data management
– Identify global resources and the control mechanism required to access them.
– Design an appropriate control mechanism for the system
– Consider how boundary conditions should be handled.
– Review and consider trade-offs.

Object-Oriented Design - OOD

Typically, there will be (at least) four different types of subsystem:

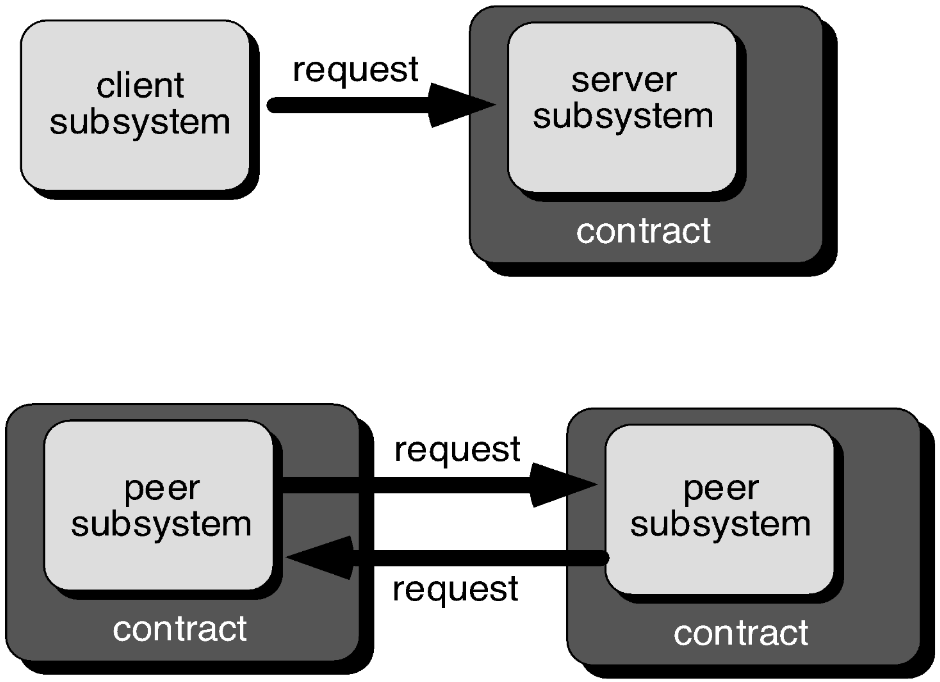
1. Problem domain subsystems: subsystems responsible for implementing customer/client requirements directly.

2. Human interaction subsystems: the subsystems that implement the user-interface (incorporating reusable GUI class libraries).

3. Task management subsystems: the subsystems that are responsible for controlling and coordinating concurrent tasks.

4. Data management subsystems: the subsystem(s) that is responsible for the storage and retrieval of objects.

Object-Oriented Design - OOD



Client/Server vs Peer-to-Peer Communication

Object-Oriented Design - OOD

When designing a subsystem, the following guidelines may be useful:

- The subsystem should have a well-defined interface through which all communication with the rest of the system occurs
- With the exception of a small number of “communication classes,” the classes within a subsystem should collaborate only with other classes within the subsystem
- The number of subsystems should be kept small
- A subsystem can be partitioned internally to help reduce complexity

Object-Oriented Design - OOD

- We also have to design individual objects and classes
- Two distinct aspects

  – A protocol description: establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs

  – An implementation description: shows implementation details for each operation implied by a message that is passed to an object. This will include:

    1. information about the object's private part
    2. internal details about the data structures that describe the object's attributes
    3. procedural details that describe operations

Object-Oriented Testing - OOT

- Begin by evaluating the correctness and consistency of the OOA and OOD models

- Recognize that the testing strategy changes

  – the concept of the 'unit' broadens due to encapsulation

  – integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario

  – validation uses conventional black box methods

- test case design draws on conventional methods (black-box testing and white-box testing) but also encompasses special features

Object-Oriented Testing - OOT

- Testing the CRC Model
  1. Revisit the CRC model and the object-relationship model
  2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition
  3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source
  4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes
  5. Determine whether widely requested responsibilities might be combined into a single responsibility.

- Steps 1 to 5 are applied iteratively to each class and through each evolution of the OOA model

Object-Oriented Testing - OOT

OOT Strategy

- Encapsulation and inheritance make testing more complicated
- Encapsulation:
  - the data members are effectively hidden and the test strategy needs to exercise both the access methods and the hidden data-structures
- Inheritance (and polymorphism):
  - the invocation of a given method depends on the context (*i.e.* the derived class for which that method is called).
  - Consequently, you need to have a new set of tests for every new context (i.e. every new derived class).
  - Multiple inheritance makes the situation even more complicated.

Object-Oriented Testing - OOT

OOT Strategy

- In conventional testing, we begin by unit testing and then proceed, incrementally, to test larger and larger sub-systems (i.e. integration testing). This approach has to be adapted for OO:

- class testing is the equivalent of unit testing:
  - operations within the class are tested
  - the state behavior of the class is examined (very often, the state of an object is persistent).
- integration testing requires three different strategies:
  - *thread-based testing*—integrates the set of classes required to respond to one input or event (incremental integration may not be possible).
  - *use-based testing*—integrates the set of classes required to respond to one use case
  - *cluster testing*—integrates the set of classes required to demonstrate one collaboration

Object-Oriented Testing - OOT

OO Test Case Design

– Each test case should be uniquely identified and should be explicitly associated with the class to be tested

– The purpose of the test should be stated

– A list of testing steps should be developed for each test and should contain:

1. a list of specified states for the object that is to be tested
2. a list of messages and operations that will be exercised as a consequence of the test
3. a list of exceptions that may occur as the object is tested
4. a list of external conditions (*i.e.,* changes in the environment external to the software that must exist in order to properly conduct the test)
5. supplementary information that will aid in understanding or implementing the test.

Object-Oriented Testing - OOT

OO Test Methods

- Random testing
  - Identify operations applicable to a class
  - Define constraints on their use
  - Identify a series of random but valid test sequences (a valid operation sequence for that class/object, *i.e.* a sequence of messages or method invocations for that class)

Object-Oriented Testing - OOT

OO Test Methods

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software (*i.e.* input and output are categorized, and test cases are designed to exercise each category)

- State-based partitioning
  - categorize and test operations based on their ability to change the state of a class

- Attribute-based partitioning
  - categorize and test operations based on the attributes that they use

- Category-based partitioning
  - categorize and test operations based on the generic function each performs

Object-Oriented Testing - OOT

OO Test Methods

– Inter-Class Testing

- For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes
- For each message that is generated, determine the collaborator class and the corresponding operator in the server object
- For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits
- For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

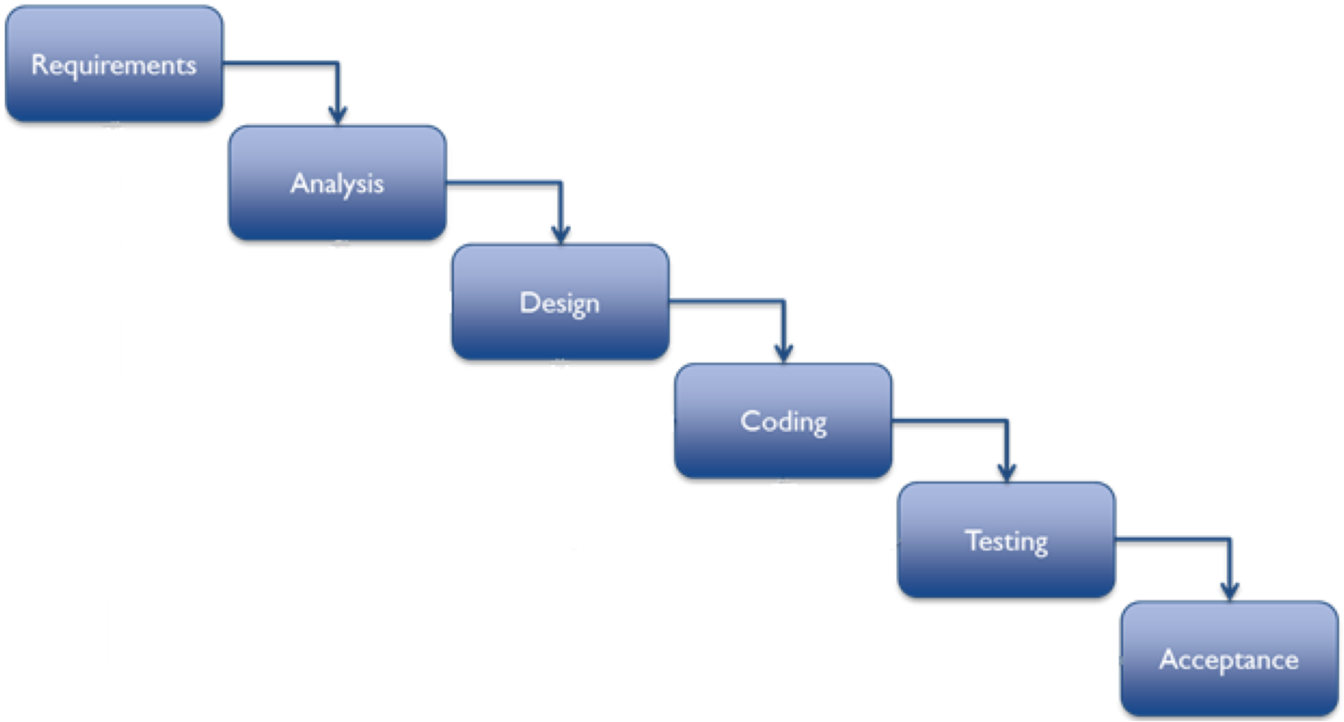
Software Process Models

Putting it all together

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



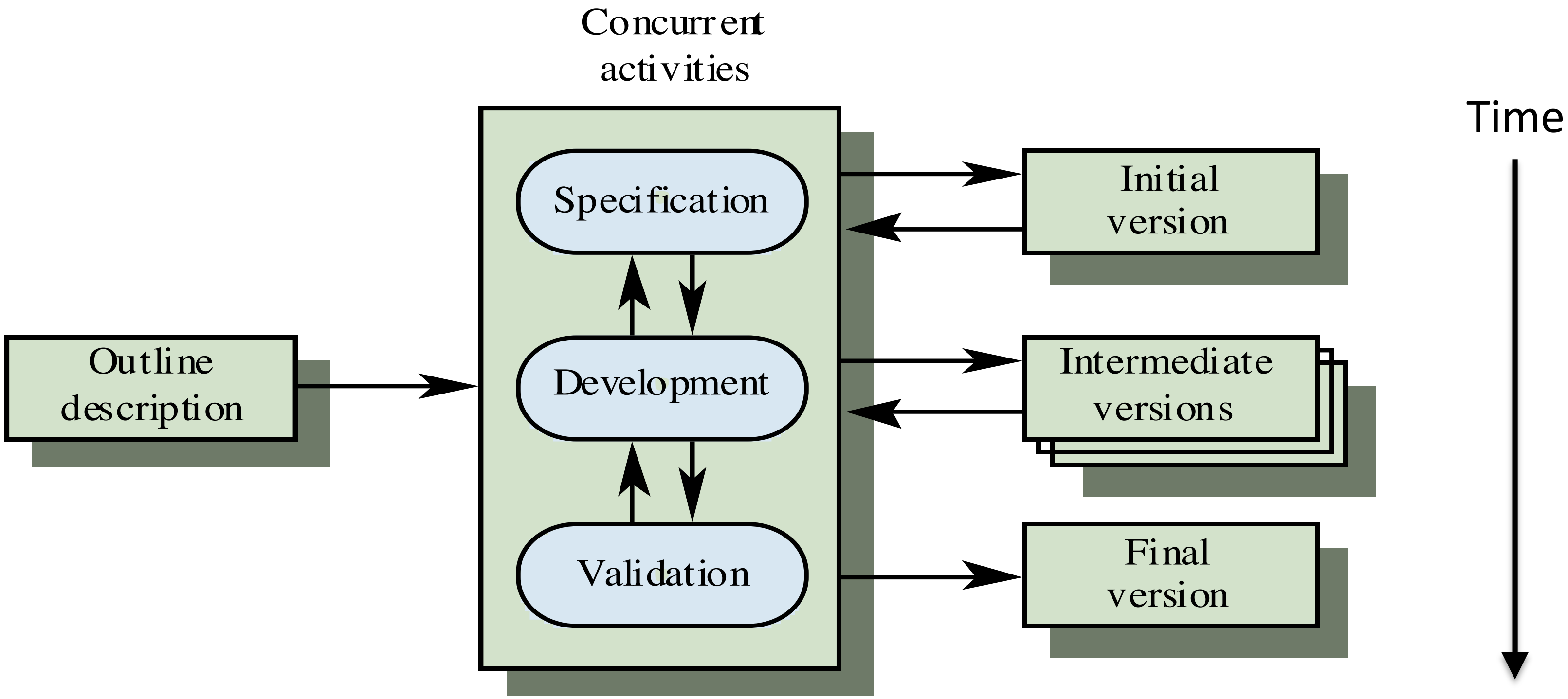
Waterfall Model

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

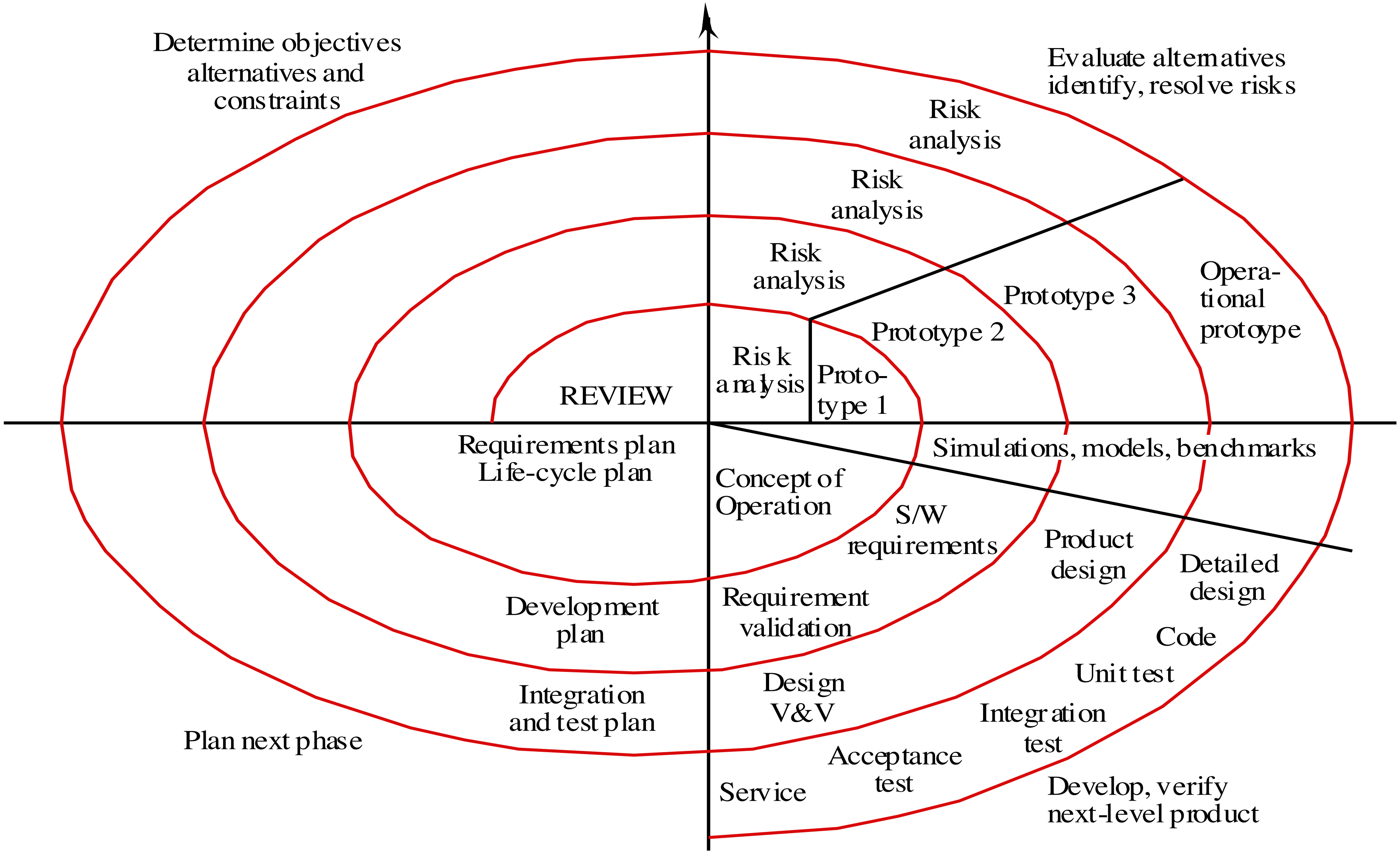
- Waterfall
  - High risk for new systems because of specification and design problems
  - Low risk for well-understood developments using familiar technology
- Prototyping (Evolutionary)
  - Low risk for new applications because specification and program stay in step
  - High risk because of lack of process visibility
- Transformational
  - High risk because of need for advanced technology and staff skills

Software Process Models

Hybrid Process Models

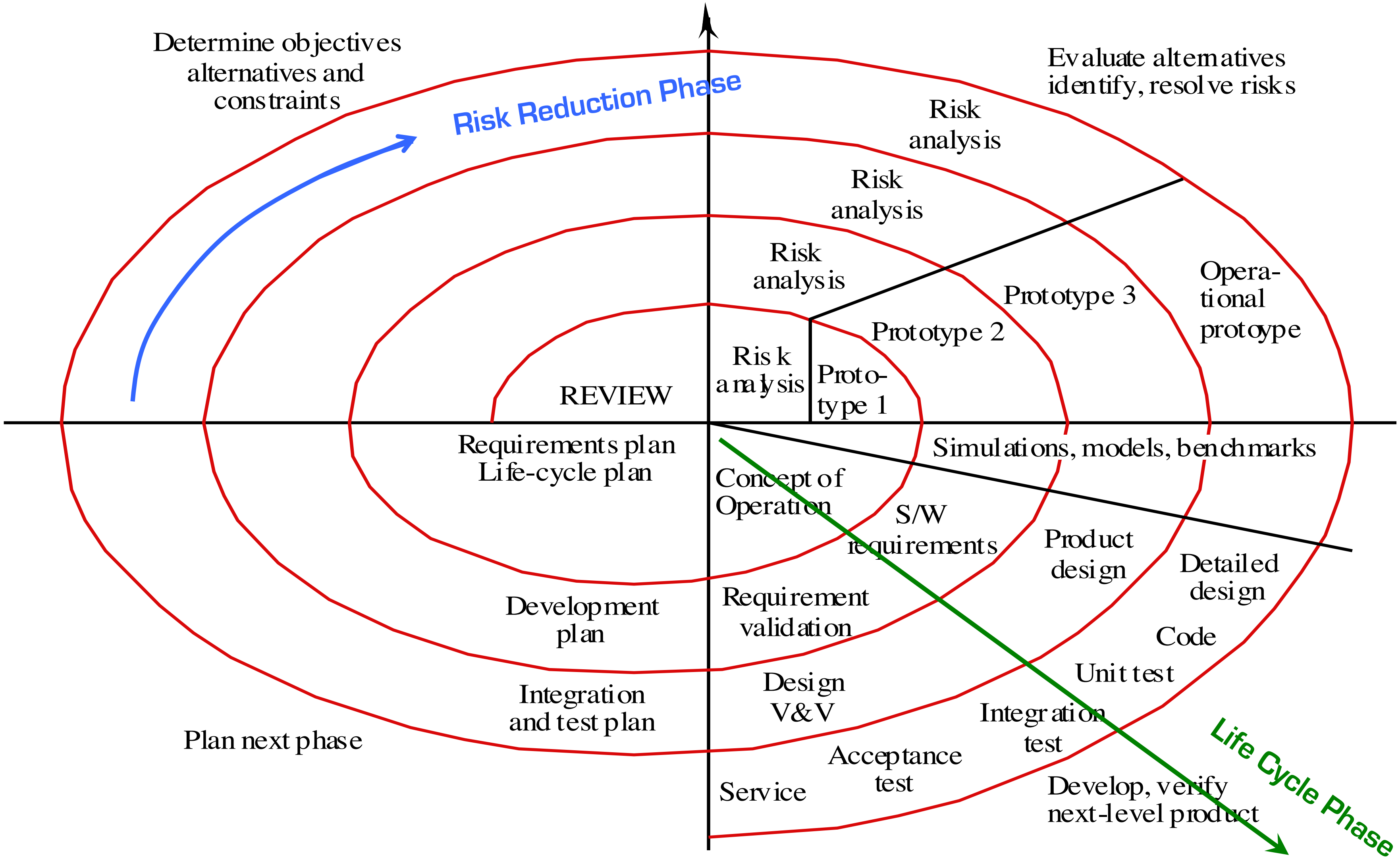
- Large systems are usually made up of several sub-systems
- The same process model need not be used for all subsystems
- Prototyping for high-risk specifications
- Waterfall model for well-understood developments

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

Process visibility

- Software systems are intangible so managers need documents to assess progress

- However, this may cause problems
  - Timing of progress deliverables may not match the time needed to complete an activity
  - The need to produce documents constrains process iteration
  - The time taken to review and approve documents is significant

- The waterfall model is still the most widely used deliverable-based model

Waterfall model documents

Activity	**Output documents**
Requirements analysis	Feasibility study, Outline requirements
Requirements definition	Requirements document
System specification	Functional specification, Acceptance test plan Draft user manual
Architectural design	Architectural specification, System test plan
Interface design	Interface specification, Integration test plan
Detailed design	Design specification, Unit test plan
Coding	Program code
Unit testing	Unit test report
Module testing	Module test report
Integration testing	Integration test report, Final user manual
System testing	System test report
Acceptance testing	Final system plus documentation

Process model visibility

Process model	**Process visibility**
Waterfall model	Good visibility, each activity produces some deliverable
Evolutionary development	Poor visibility, uneconomic to produce documents during rapid iteration
Formal transformations	Good visibility, documents must be produced from each phase for the process to continue
Reuse-oriented development	Moderate visibility, it may be artificial to produce documents describing reuse and reusable components.
Spiral model	Good visibility, each segment and each ring of the spiral should produce some document.

Software Development
vs.
Software Engineering

Software Engineering

Software engineering is the branch of systems engineering concerned with the development of large and complex software-intensive systems

Software Engineering

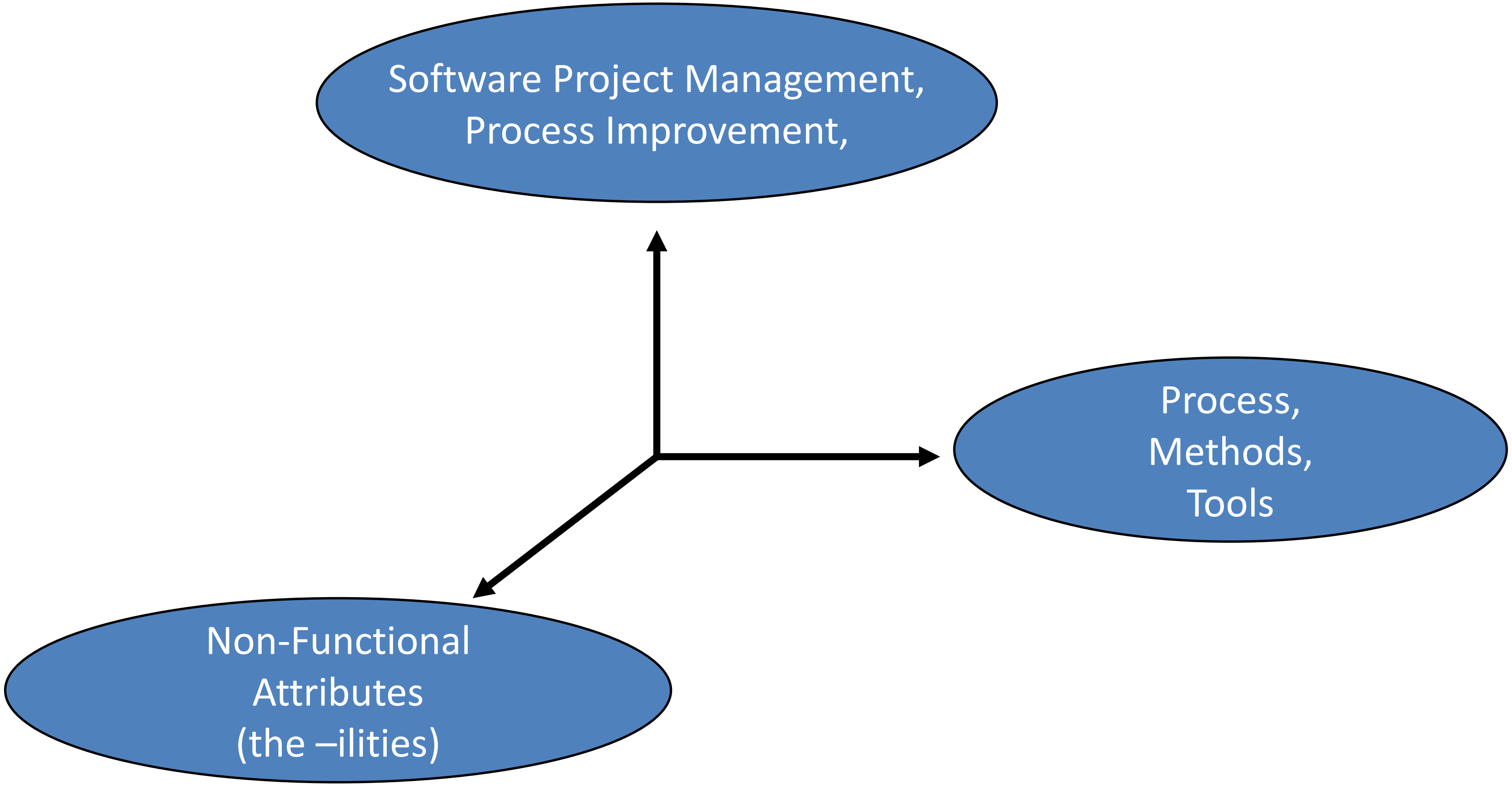
It is concerned with the:

- Processes
- Methods
- Tools

for the development of software intensive systems in an *economic* and *timely* manner.

A. Finelstein and J. Kramer, "Software Engineering: A Road Map" in
"The Future of Software Engineering" , Anthony Finkelstein (Ed.), ACM Press 2000

Software Engineering



The Three Dimensions of Software Engineering

Software Engineering

- People
  - People matter
  - Software engineering is as much about the organization and management of people as it is about technology
  - People use the system
  - People design the system
  - People build the system
  - People maintain the system
  - People pay for the system

- Product

- Process

Software Development vs. Software Engineering

- Software textbooks tend to emphasize the management aspects and process aspects of software development

- While software engineering is certainly important, it is not everything

- The following points, taken from J. A. Whittaker and S. Atkin, “Software Engineering Is Not Enough”, IEEE Software, July/August 2002, pp. 108-115.

Software Development vs. Software Engineering

Software Development Is More Than Methodology (or Process, or Estimation, or Project Management)

- 'Software development is a fundamentally technical problem for which management solutions can be only partially effective.'
- Coding is immensely difficult without a good design but still very difficult with one
- Maintaining code is next to impossible without good documentation and formidable with it.'

Software Development vs. Software Engineering

Programming is Hard

- ‘Programming remains monstrously complicated for the vast majority of applications’

- ‘The only programs that are simple and clear are the ones you write yourself’

- (Why?)

Software Development vs. Software Engineering

Documentation is Essential

- ‘There is rarely such a thing as too much documentation …
- Document Control Blocks and Data Structures

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

Software Development vs. Software Engineering

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’

Software Development vs. Software Engineering

Failure is Inevitable – You Need To Handle It

– Constraints prevent failure

– Exceptions let the failure occur and then trap it (and handle it with a special routine called an exception handler)

– ‘Failure recovery is often difficult.’

Software Development vs. Software Engineering

Before you can even begin, you must be an expert in both the problem domain and the solution domain

- Problem-domain expertise
  (understanding and modelling the problem)

- Solution-domain expertise
  (editors, compilers, linkers, and debuggers, … make-utilities, runtime libraries, development environments, version-control managers, and … the operating system)

- **'Developers must be masters of their programming language and their OS; methodology alone is useless.'**