Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques

Software Engineering Process

T. Cioara, C. Pop, V. Chifu
2025

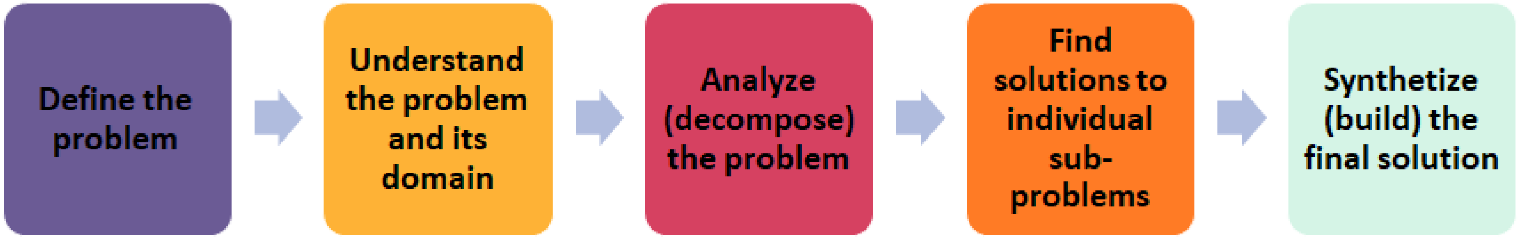
Problem Solving

- What do all have in common?

© 2004 Buena Vista Pictures

Problem Solving

- Main steps



Define the problem

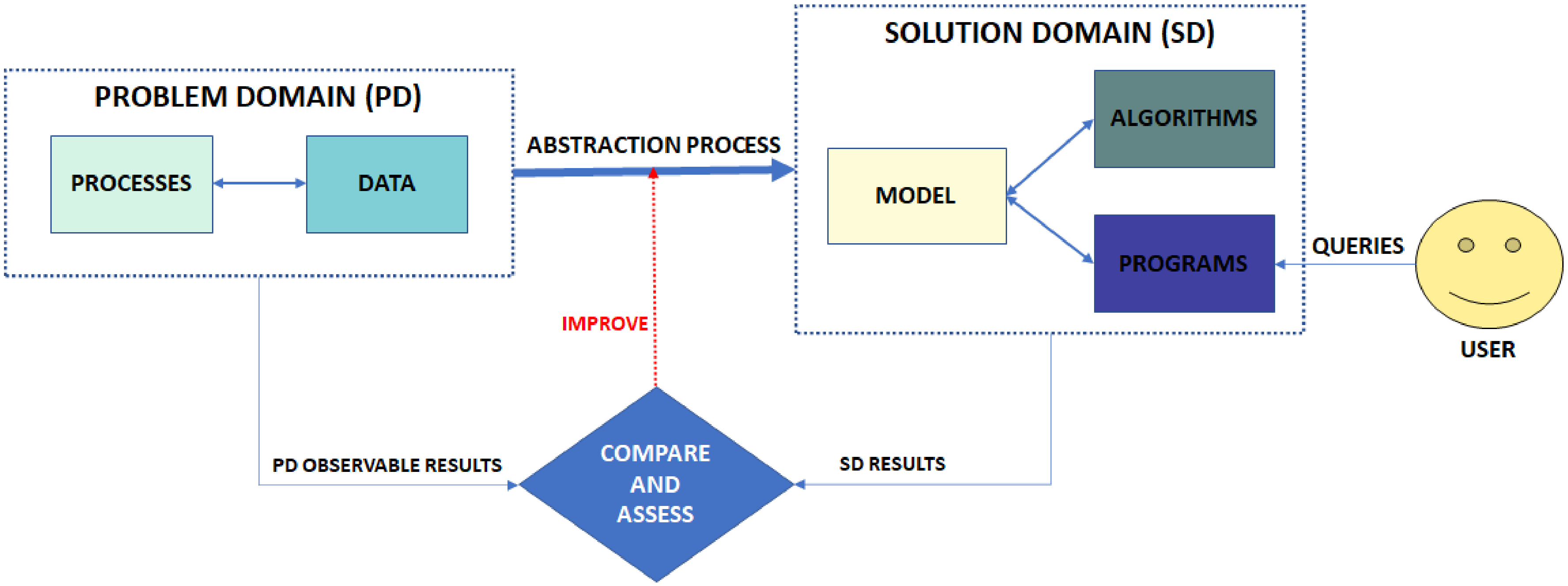
Understand the problem and its domain

Analyze (decompose) the problem

Find solutions to individual sub-problems

Synthetize (build) the final solution

- Problem Domain and Solution Domain



PROBLEM DOMAIN (PD)

PROCESSES

DATA

ABSTRACTION PROCESS

SOLUTION DOMAIN (SD)

MODEL

ALGORITHMS

PROGRAMS

QUERIES

USER

IMPROVE

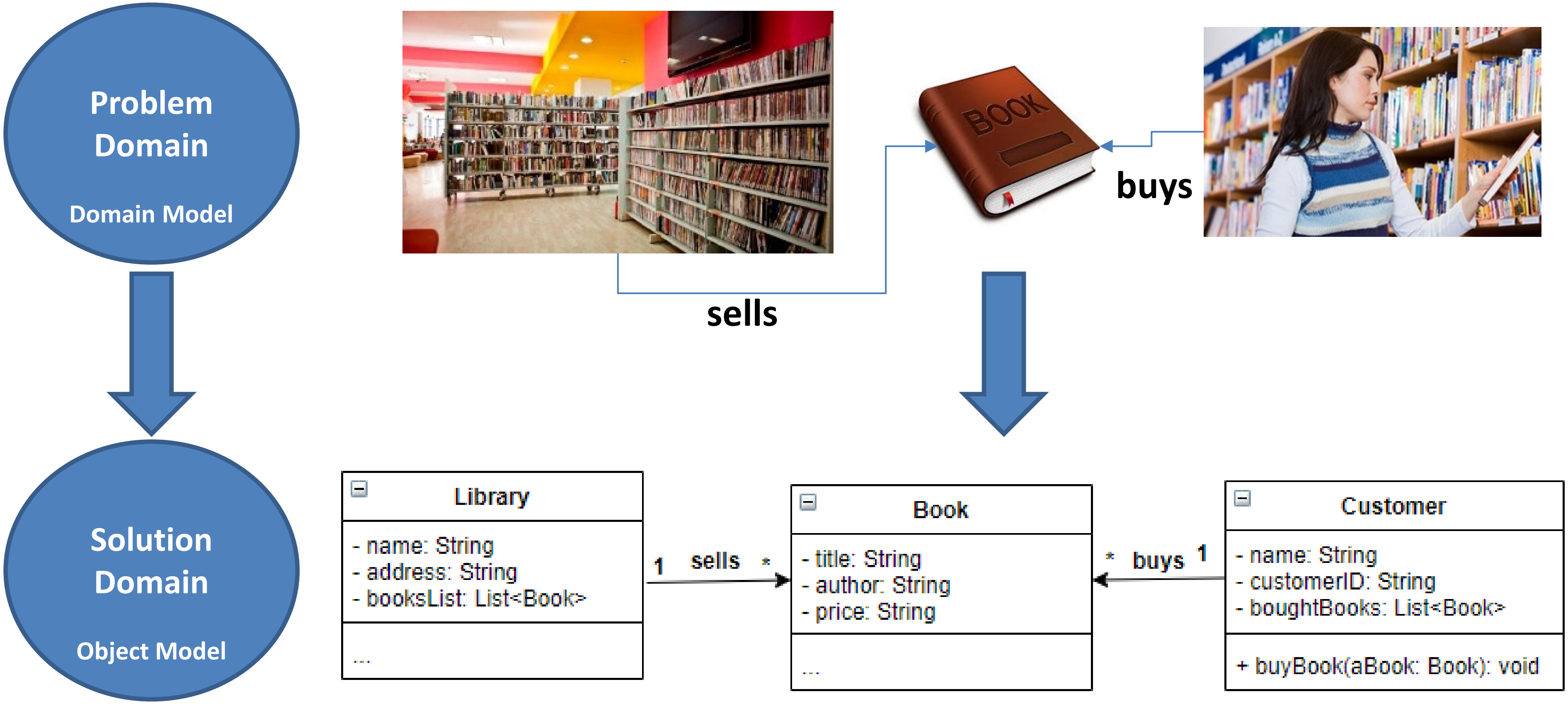
PD OBSERVABLE RESULTS

COMPARE AND ASSESS

SD RESULTS

Problem Solving

- **Modeling the semantic gap – an object-oriented approach**



Problem Domain

Domain Model

sells

buys

Solution Domain

Object Model

Library

- name: String
- address: String
- booksList: List<Book>

...

1 sells *

Book

- title: String
- author: String
- price: String

...

* buys 1

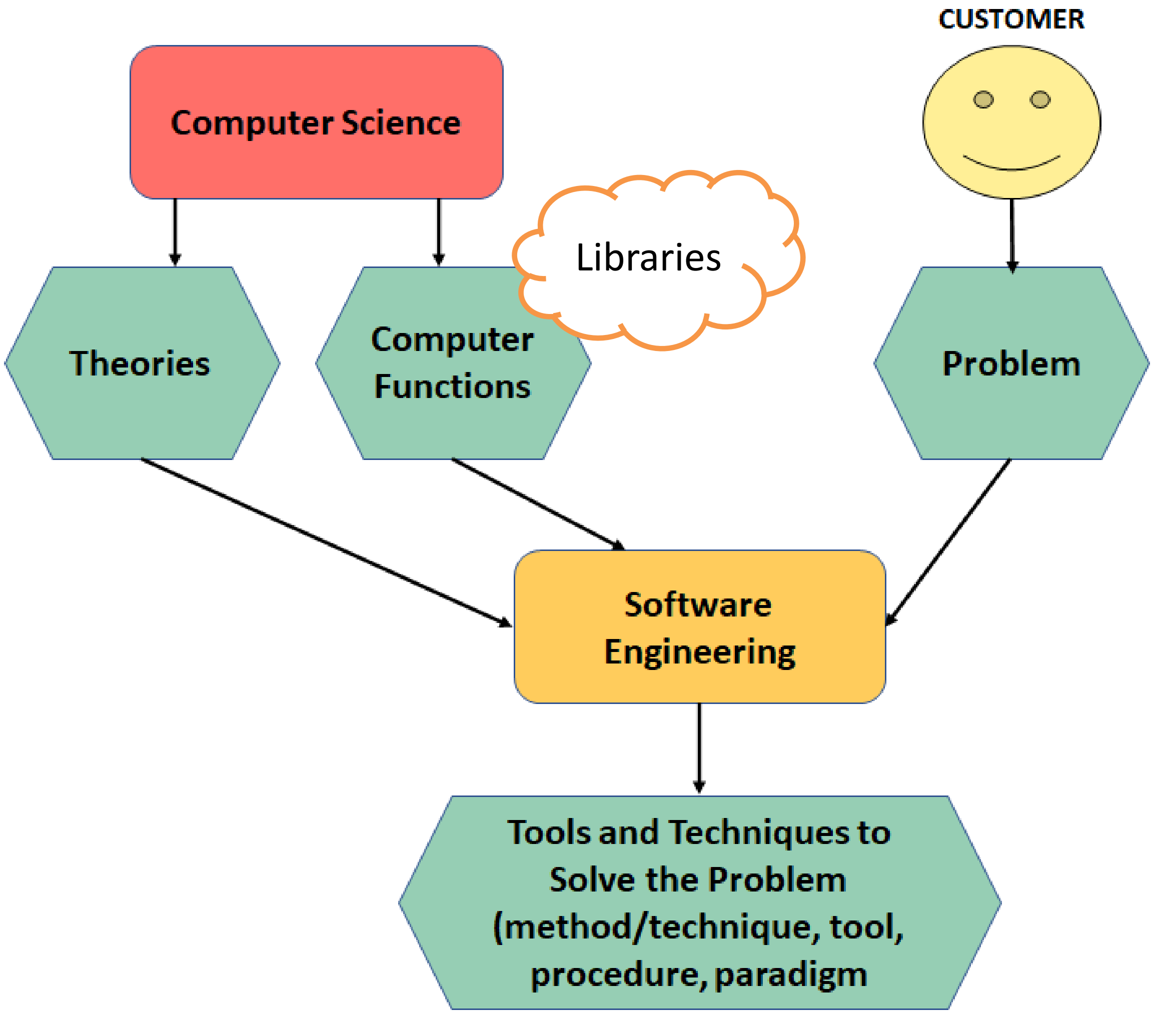
Customer

- name: String
- customerID: String
- boughtBooks: List<Book>

+ buyBook(aBook: Book): void

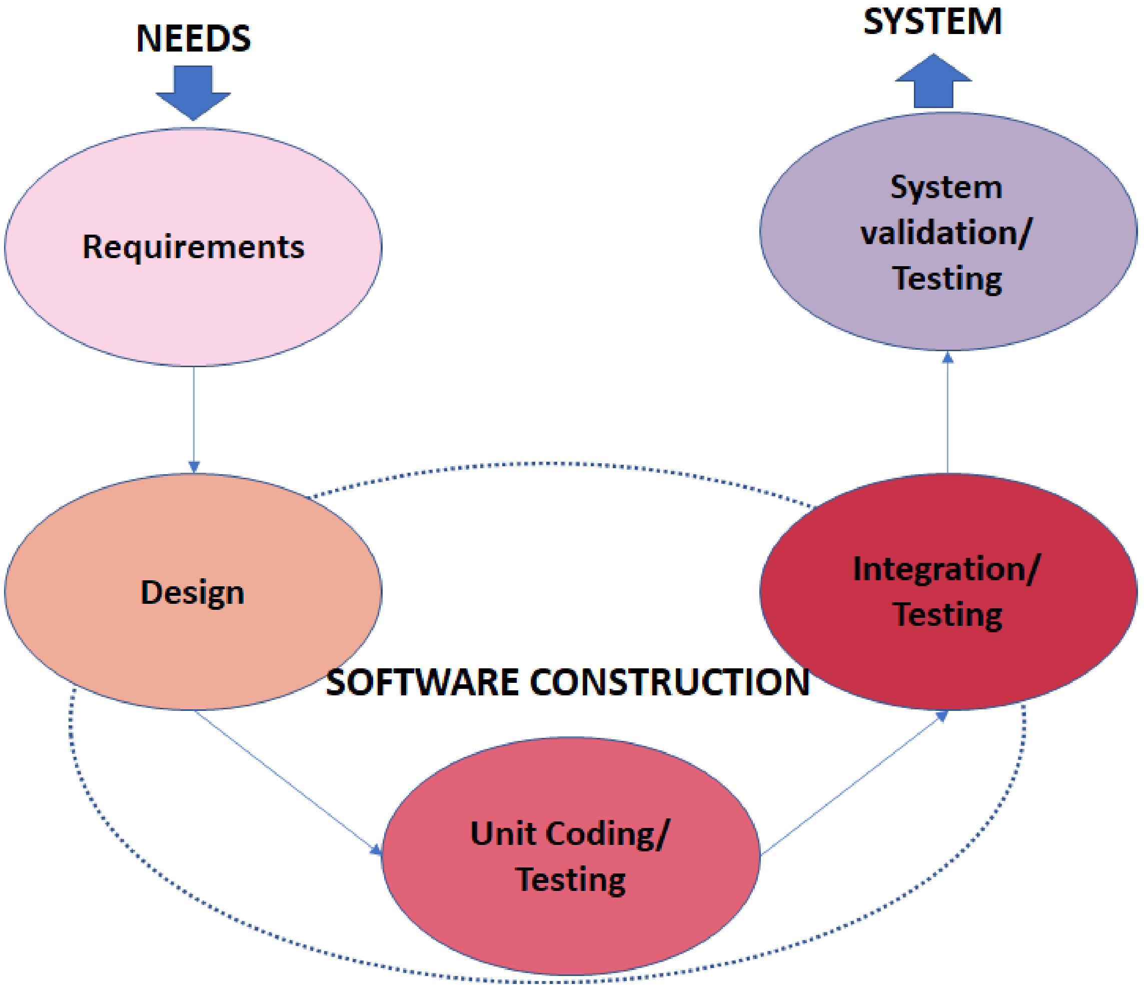
Adapted from [Source]

Software Engineering


CUSTOMER
Computer Science
Libraries
Theories
Computer Functions
Problem
Software Engineering
Tools and Techniques to Solve the Problem (method/technique, tool, procedure, paradigm

Adapted from [Source]

Software engineering process



Principles

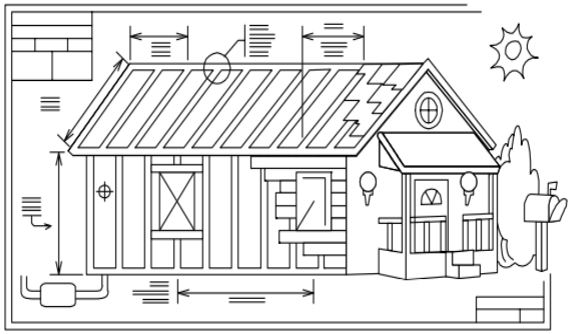
- Provide value to users
- KISS
- Keep the vision
- Plan ahead and reuse
- Think!

Goal

- High quality

Software engineering process

- Understanding of software development

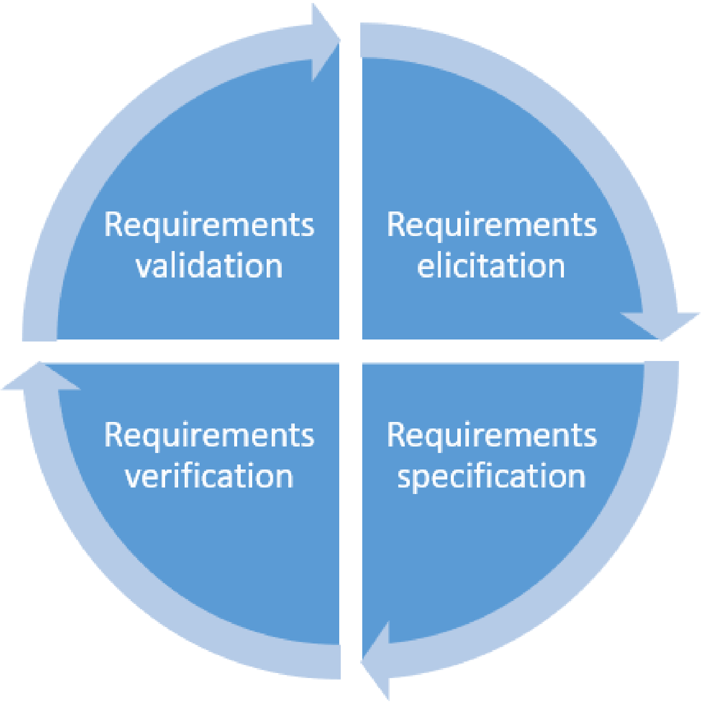


[Source]

Building a house analogy	
Determine and analyze the residents' requirements	Architect/ engineer
Produce and document the overall design of the house	
Produce detailed specifications	
Identify and design the components	
Build each component of the house	Worker / programmer
Test each house component	
Integrate the components and test as a whole	
Make final modifications after the residents have moved in	
Continue the maintenance by the residents	

Software engineering process

Requirements Engineering

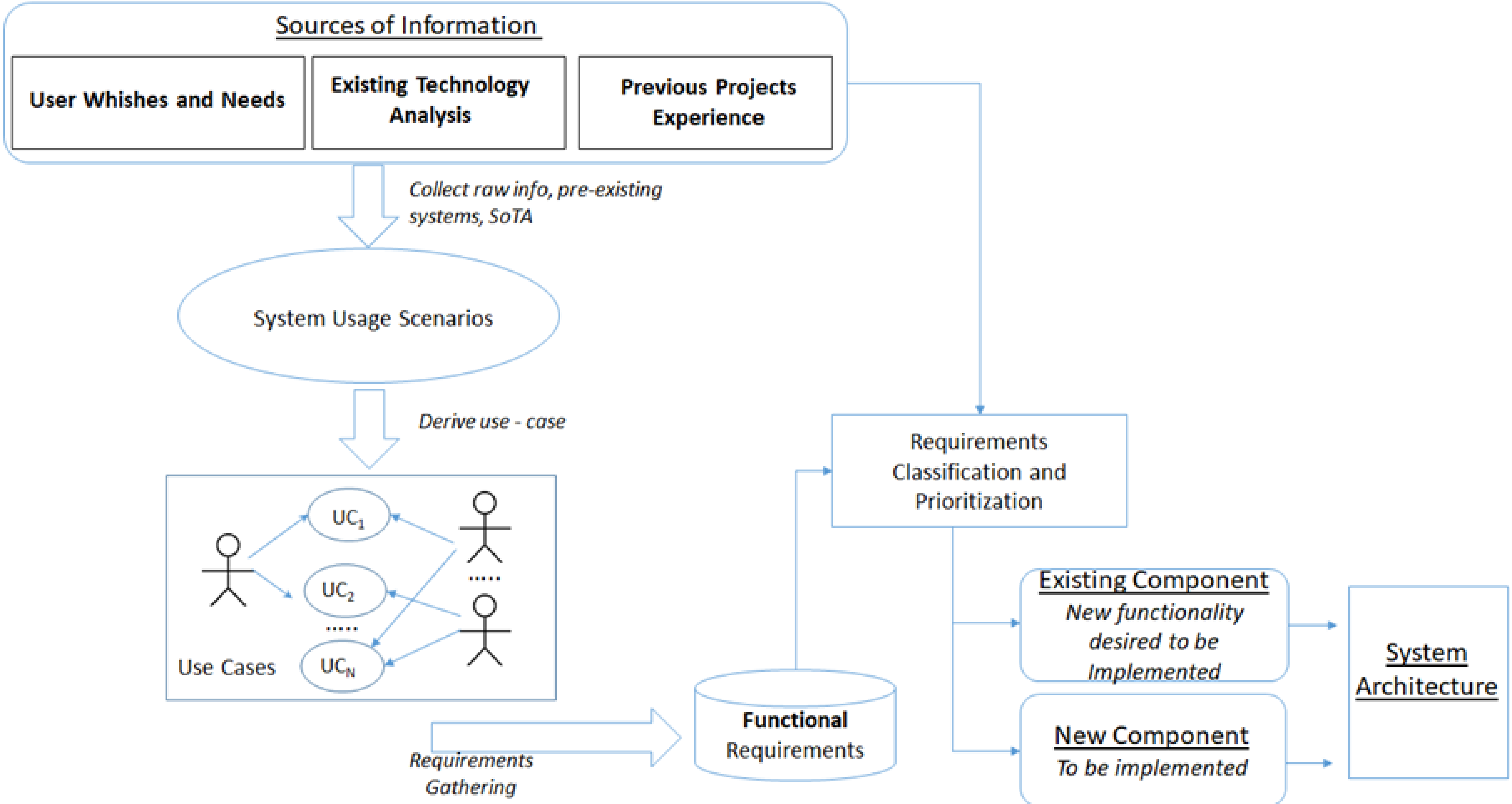


- Types of requirements: functional, non-functional, constraints

Pay attention to requirements to minimize the changes to a system after development begins!

Software engineering process
Requirements Engineering

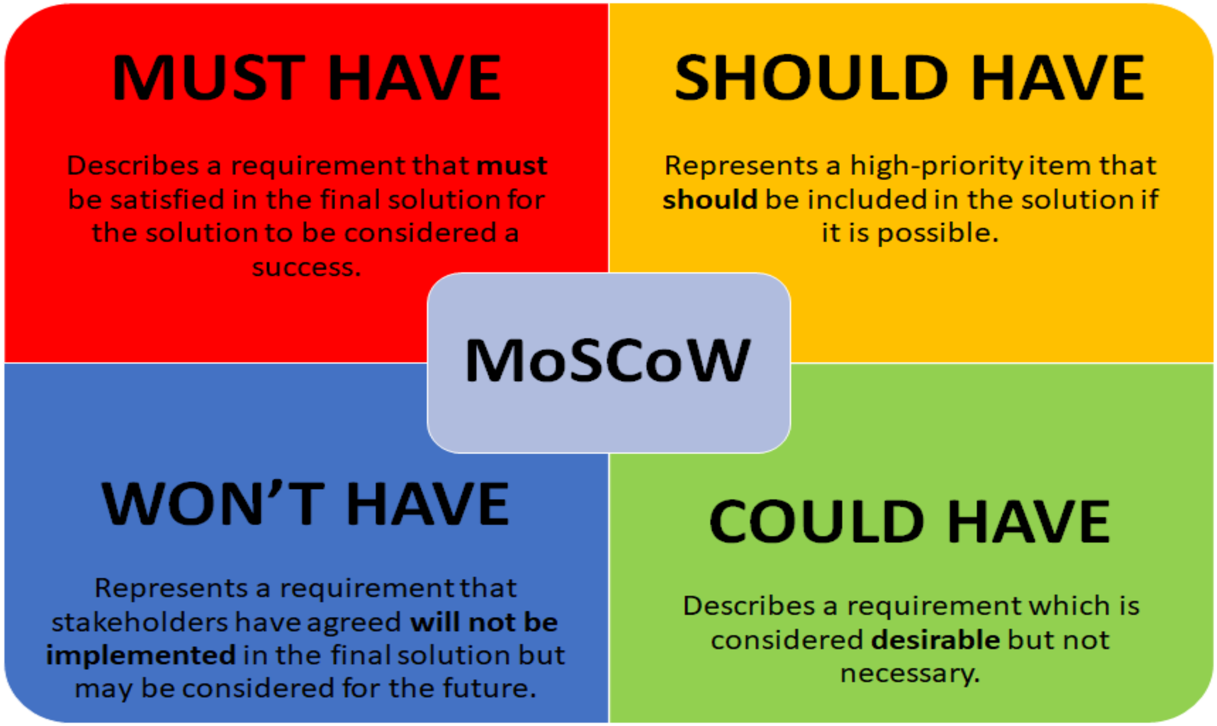
- Functional requirements and system architecture



Software engineering process

Requirements Engineering

- Requirements Prioritization and Classification – the MoSCoW technique



MoSCoW

MUST HAVE

Describes a requirement that **must** be satisfied in the final solution for the solution to be considered a success.

SHOULD HAVE

Represents a high-priority item that **should** be included in the solution if it is possible.

WON'T HAVE

Represents a requirement that stakeholders have agreed **will not be implemented** in the final solution but may be considered for the future.

COULD HAVE

Describes a requirement which is considered **desirable** but not necessary.

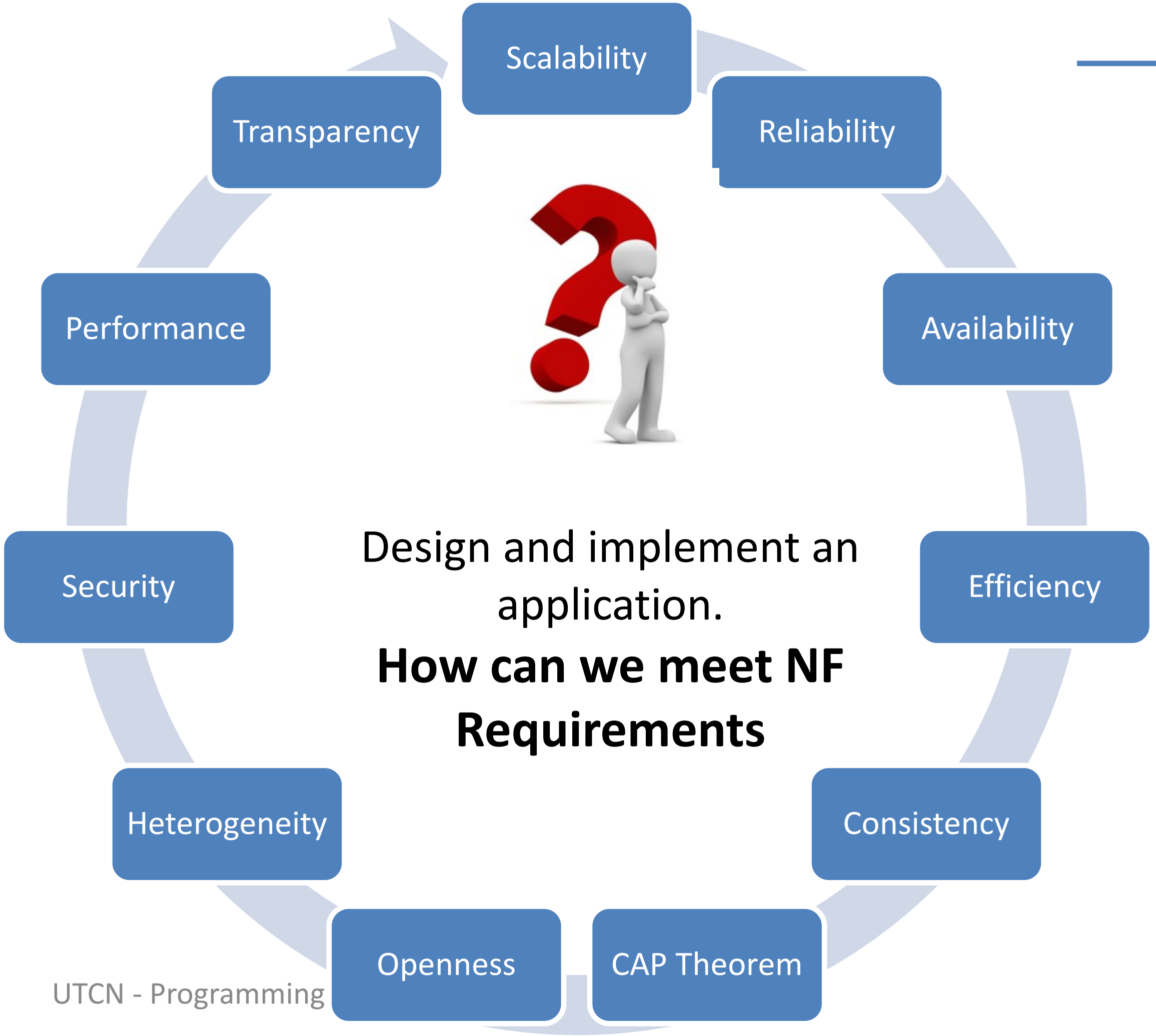
- Trade-offs in software development

Adapted from [Source]

Software engineering process

Requirements Engineering

- Nonfunctional requirements



Design and implement an application.

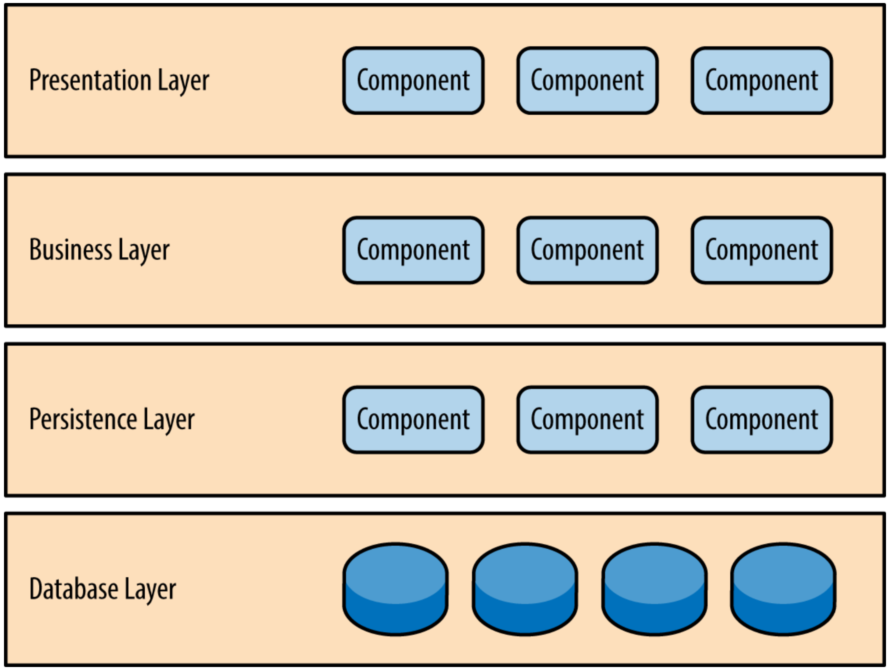
How can we meet NF Requirements

NF in practice
SLI - Service Level Indicators
SLO – Service Level Objectives
SLA - Service Level Agreements

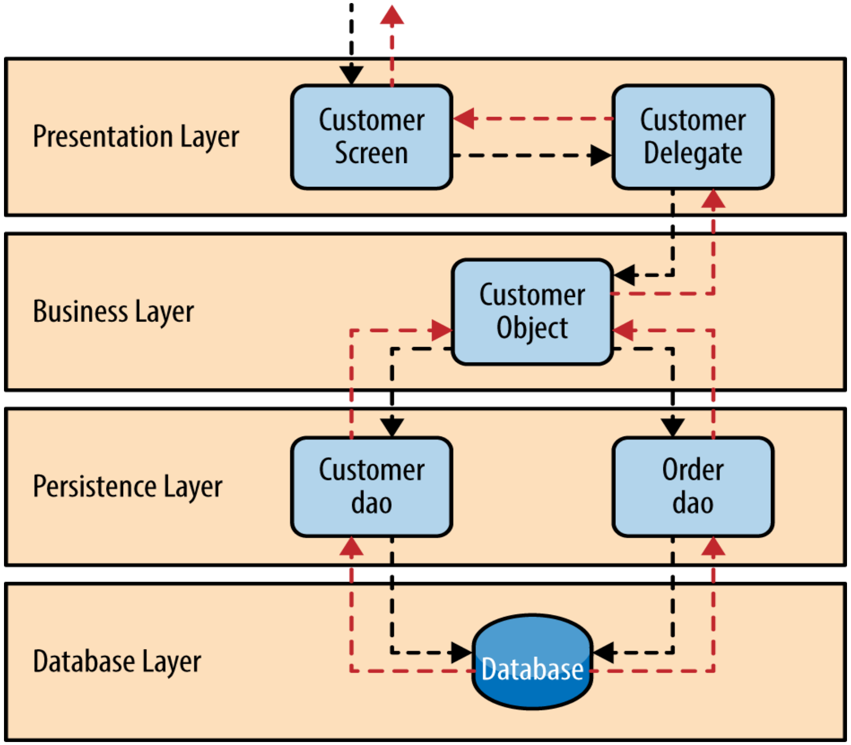
Software engineering process

High level system architecture design

LAYERED ARCHITECTURE



EXAMPLE



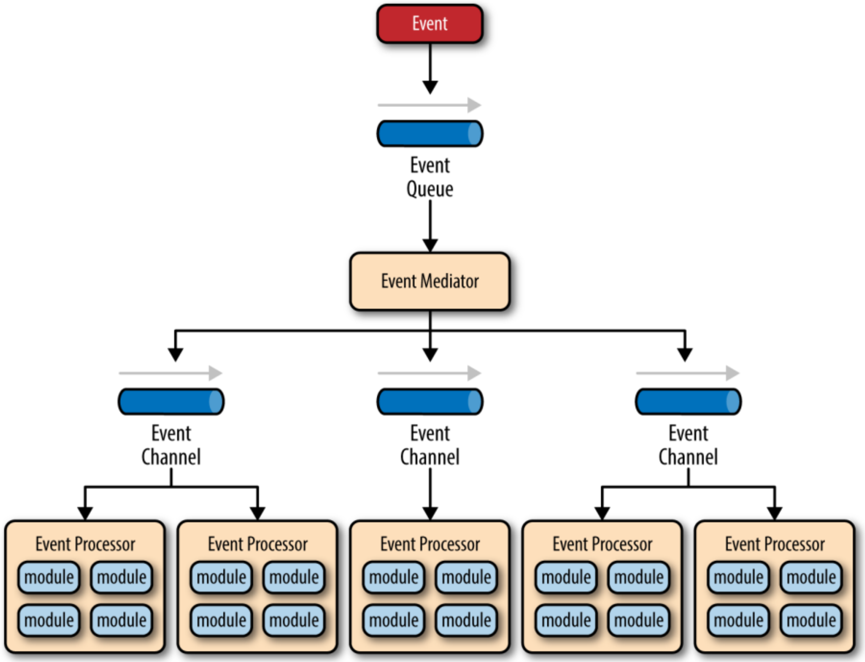
[Source]

Software engineering process

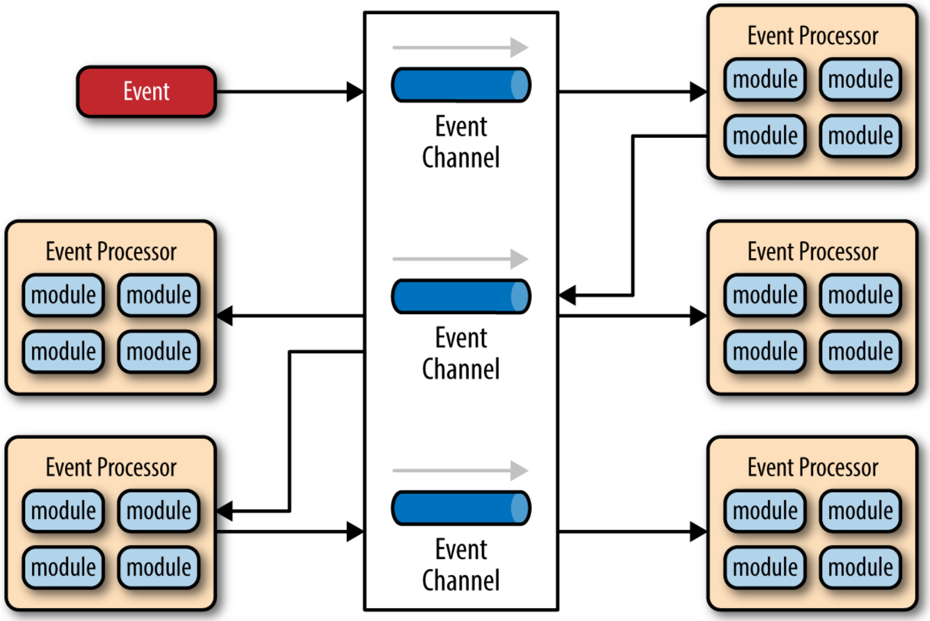
High level system architecture design

- Event driven architecture
  - Asynchronous pattern based on highly decoupled, event processing components

MEDIATOR TOPOLOGY



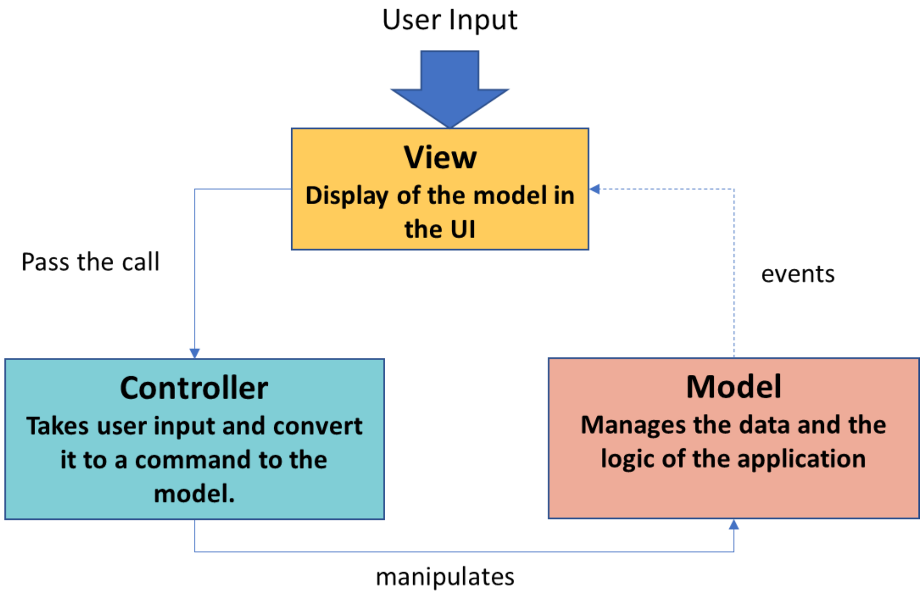
BROKER TOPOLOGY



[Source]

Software engineering process
High level system architecture design

Model View Controller



User Input

View
Display of the model in the UI

Pass the call

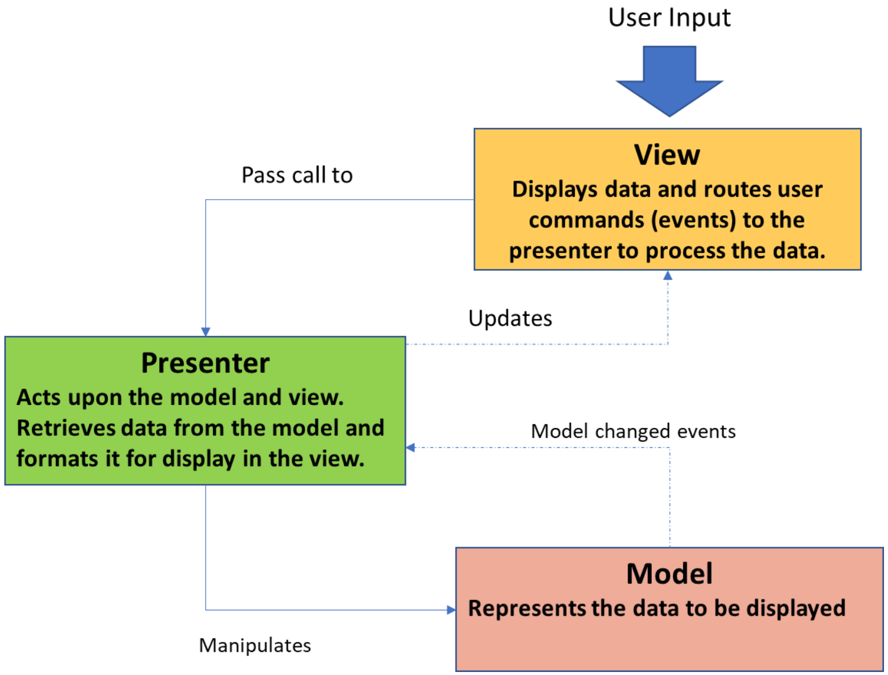
events

Controller
Takes user input and convert it to a command to the model.

Model
Manages the data and the logic of the application

manipulates

Model View Presenter



User Input

View
Displays data and routes user commands (events) to the presenter to process the data.

Pass call to

Updates

Presenter
Acts upon the model and view. Retrieves data from the model and formats it for display in the view.

Model changed events

Model
Represents the data to be displayed

Manipulates

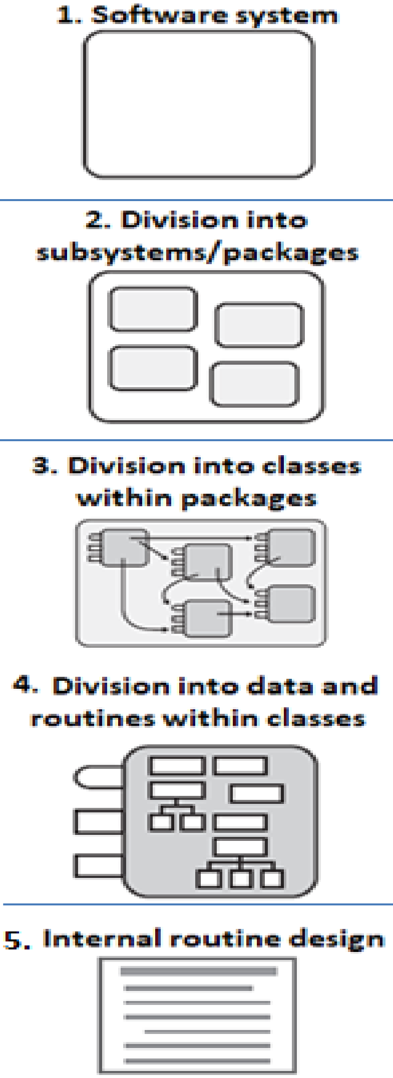
- Efficient for code re-use and parallel development

Software engineering process
Detailed Design

- Objective: Obtain a good (sub)system / component
  - (partial) reuse existent resources;
  - (partial) new resources
- Design
  - iterative try-error and divide and conquer type activity
- Breaking complexity
  - top-down or bottom-up approaches
- Skilled designers
  - Use heuristics
  - Make trade offs

Software engineering process

Detailed Design


1. Software system
2. Division into subsystems/packages
3. Division into classes within packages
4. Division into data and routines within classes
5. Internal routine design

[Source]

Best practices

- Minimize the relationships between subsystems;
- Avoid circular relationships
- Consider common design patterns

- OOP Paradigms, class level design, design patterns, SOLID principles, etc.
- Loose coupling, high cohesion

- Code writing rules, clean code

Software engineering process
Detailed Design

- Desirable Software Features
  - **External features** – visible to the customer
    - Availability, reliability, performance, fault tolerant, maintainability, security, interoperability, portability, usability, functionality, system integrity, efficiency.
  - **Internal features** – perceptible only to developers
    - minimal complexity, ease of maintenance, extensibility, reusability, high fan-in, low-to-medium fan-out, stratification, standard techniques

External features count in the end, but internal features make it possible to obtain them.

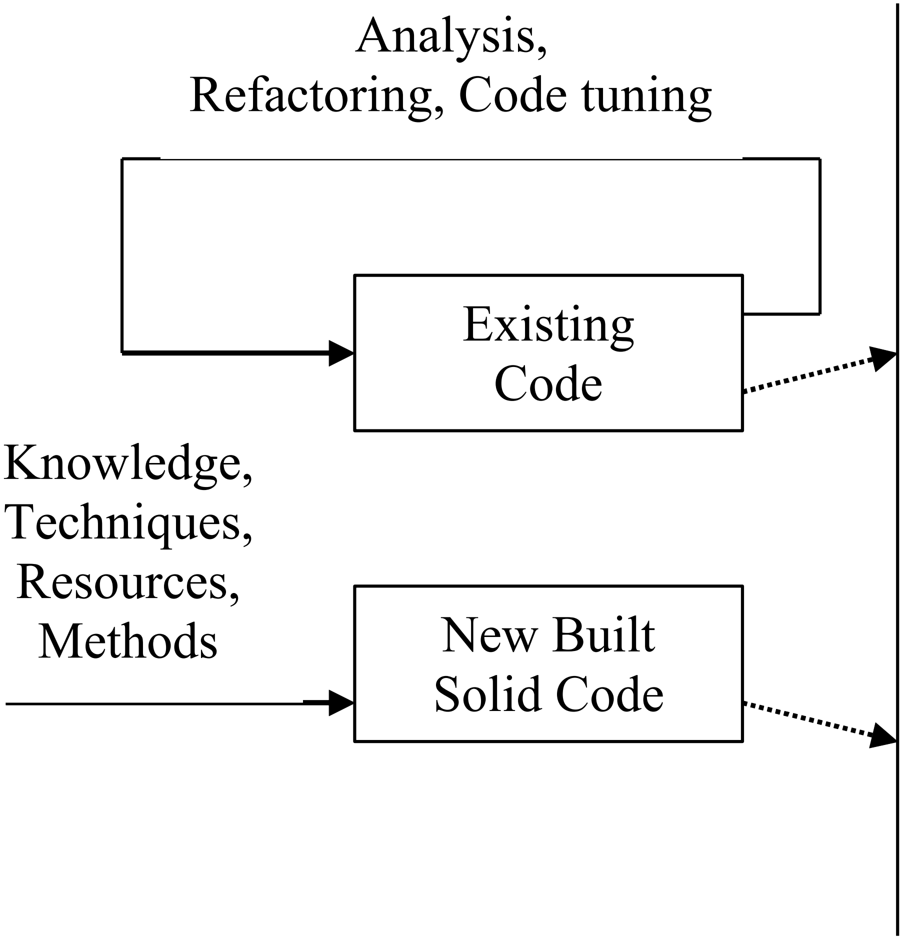
Software engineering process

Detailed Design

- Heuristics
  - Design is nondeterministic => skillful application of an effective set of heuristics is the core activity
    - Find real-world objects
    - Build abstractions and hierarchies of abstractions
    - Encapsulate implementation details
    - Inherit or delegate
    - Keep coupling loose and coherence high
    - Look for common design patterns
    - Think of associating things
    - Design for test
    - Draw diagrams
    - Keep the design modular

Software engineering process

Implementation



Analysis,
Refactoring, Code tuning

Existing
Code

Knowledge,
Techniques,
Resources,
Methods

New Built
Solid Code

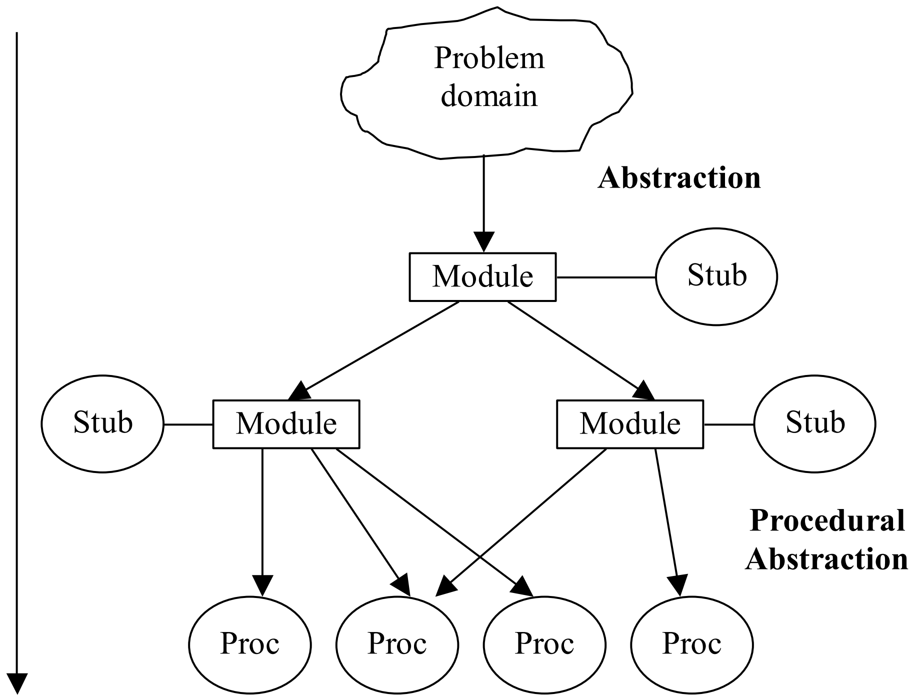
Features

Reliability
Performance
Failure tolerant
Maintenance
Portability
Interoperability
Security
Testability

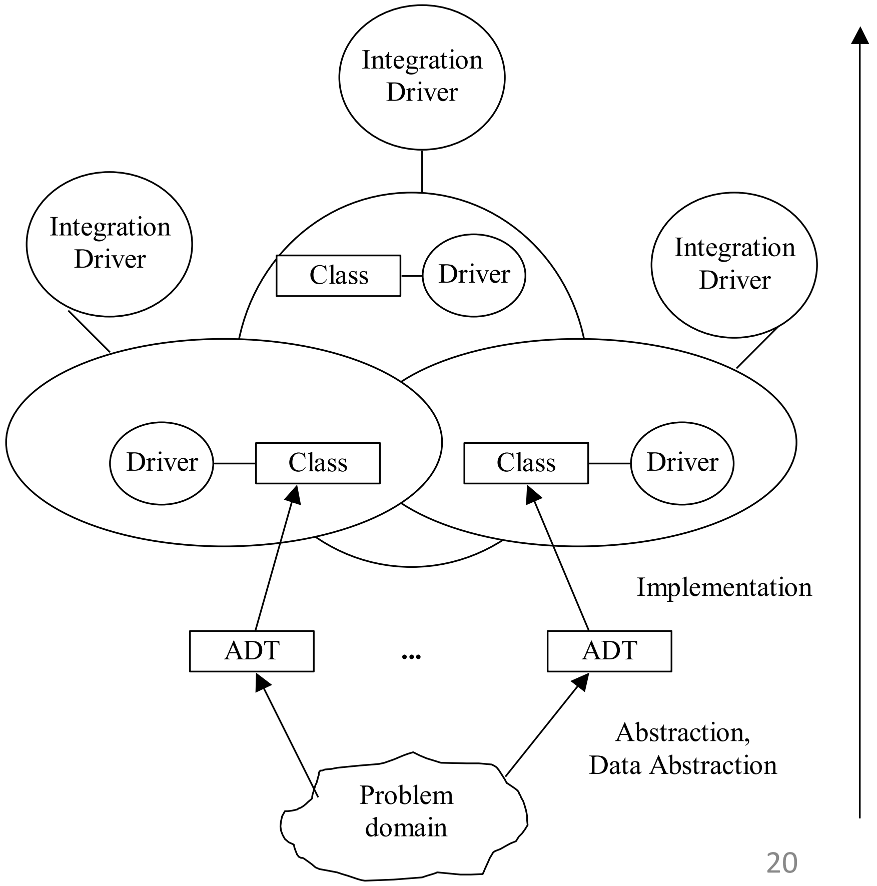
Reusable
Extensible / Scalable
Low complexity
Loose coupling
Fan-in / fan-out
Completeness
Standardized

Software engineering process
Implementation

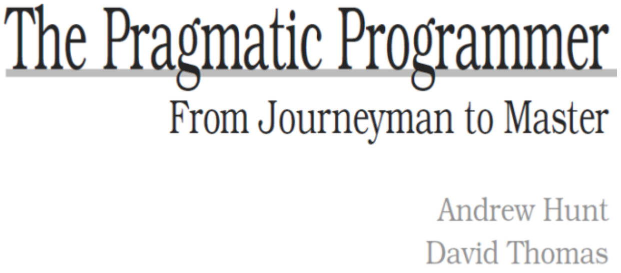
Divide et Impera - Top Down
(Procedural Development)



Divide et Impera – Bottom Up
(OO development))



Software engineering process
The Pragmatic Software Engineer


The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt
David Thomas

Characteristics

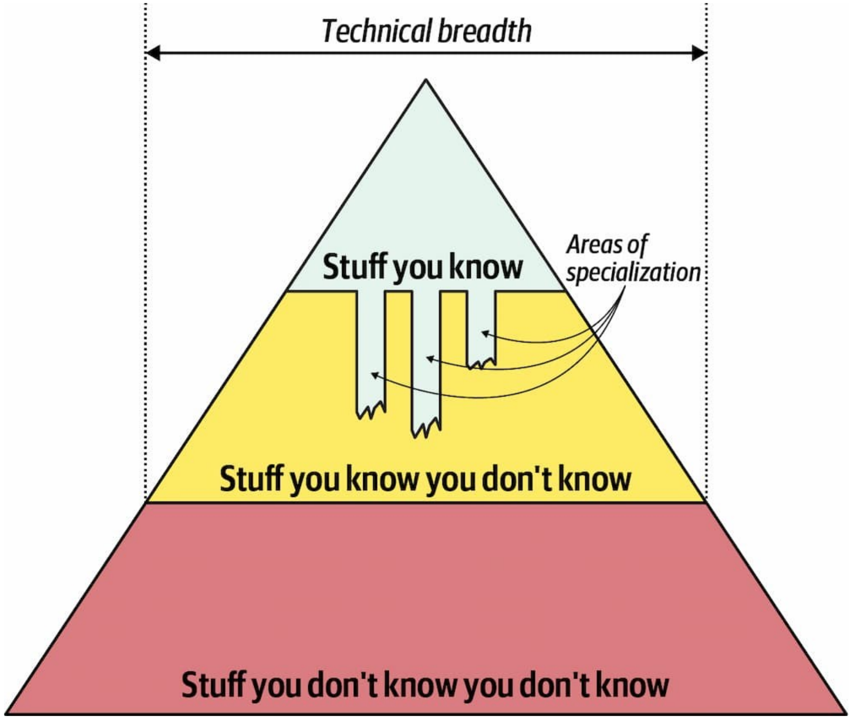
- Early adopter/fast adapter
- Tend to ask questions
- Critical thinker
- Realistic
- Try hard to be familiar with a broad range of technologies

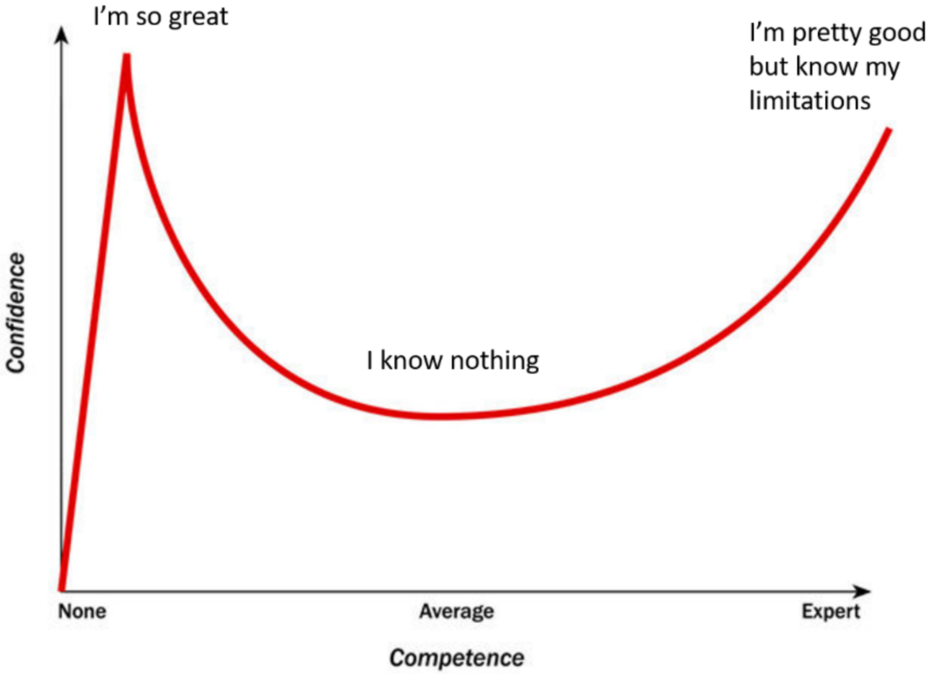
TIP1: Care about your CODE

TIP2: Think! About Your Work

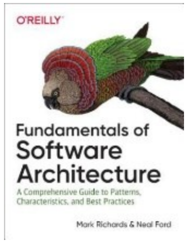
Software engineering process

The Pragmatic Programmer


Technical breadth
Stuff you know
Areas of specialization
Stuff you know you don't know
Stuff you don't know you don't know



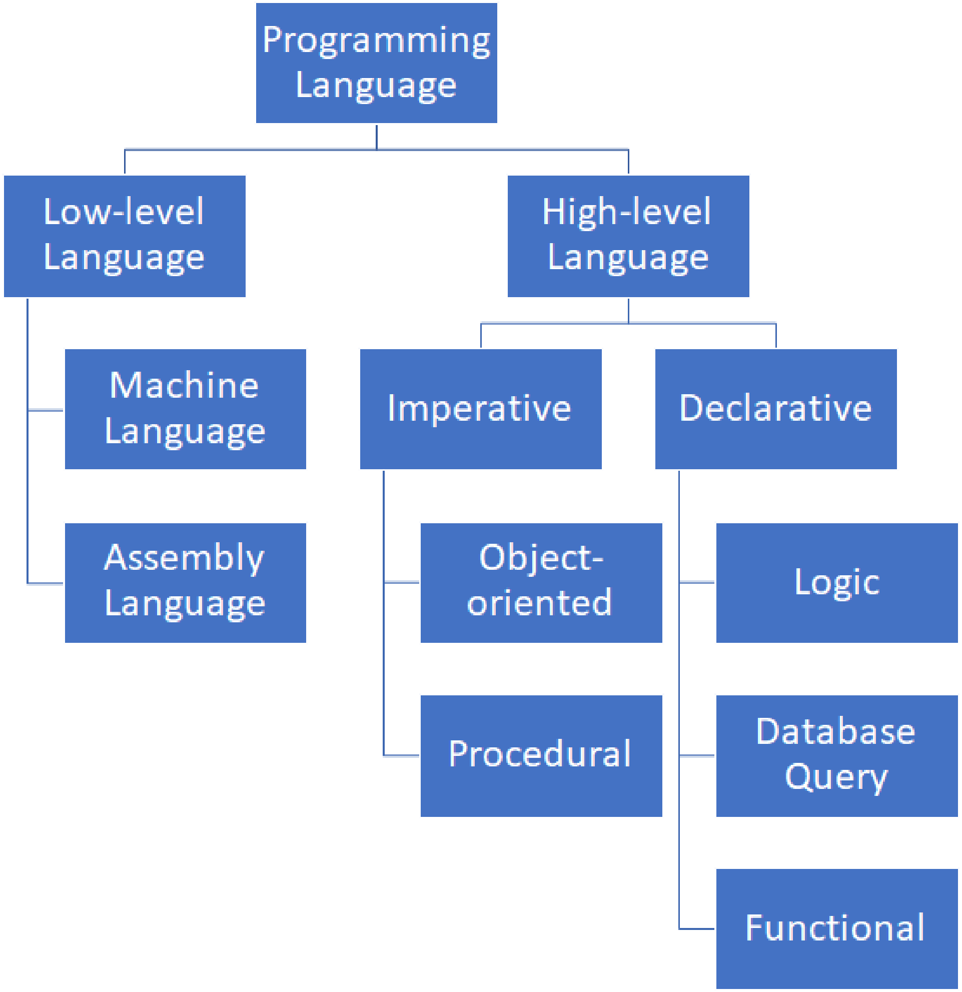
Fundamentals of Software Architecture: An Engineering Approach by Mark Richards


O'REILLY
Fundamentals of Software Architecture
A Comprehensive Guide to Patterns, Characteristics, and Best Practices
Mark Richards & Neal Ford

Software engineering process

Implementation – Programming Languages

- **Choosing the right programming language**
  - Difficult task
  - Decision metrics
    - Ease of learning,
    - ease of understanding,
    - speed of development,
    - portability,
    - fit-for-purpose



Many languages are multi-paradigm!

Software engineering process

Implementation – Programming Languages

Imperative Programming

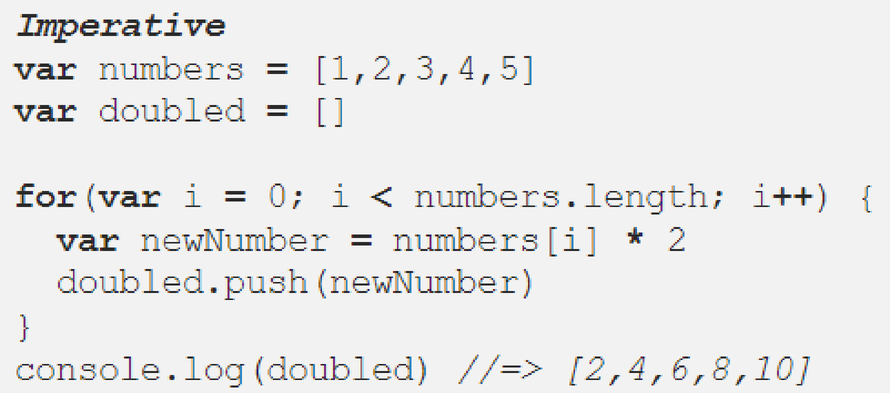
- Uses statements that change a program’s state
- Every step of the program is executed sequentially
- Sub-categories
  - Procedural: relies on procedures, algorithms - ex. Pascal, C++
  - Object-oriented: relies on classes, objects containing data and methods (procedures) – ex. C++, Java, C#
- Java is imperative and object oriented

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> output = new ArrayList<>();
for (Integer x : input) {
     if (x % 2 == 0) {
        output.add(x);
     }
}
```

Software engineering process
Implementation – Programming Languages

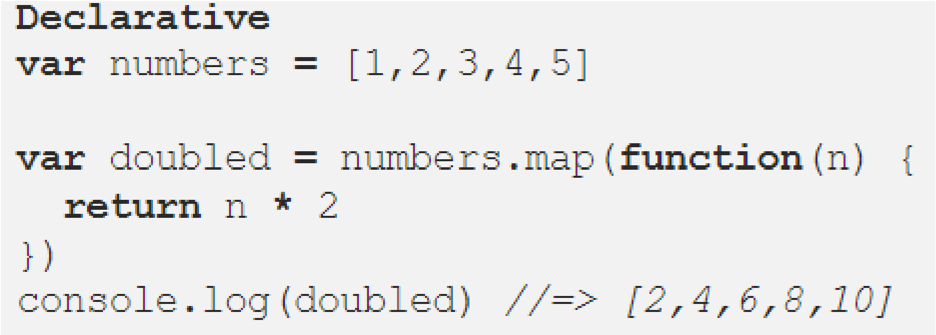
Declarative Programming

- The programmer declares/describes the properties of the result but not actually how to compute the result
- Sub-categories
  - Functional: relies on functions, Lambda Calculus – e.g., Lisp. ML, Haskel
  - Logic: uses predicate calculus – e.g., Prolog
  - Database Query: e.g., SQL
- Imperative paradigm versus declarative paradigms



```
Imperative
var numbers = [1,2,3,4,5]
var doubled = []

for(var i = 0; i < numbers.length; i++) {
  var newNumber = numbers[i] * 2
  doubled.push(newNumber)
}
console.log(doubled) //=> [2,4,6,8,10]
```



```
Declarative
var numbers = [1,2,3,4,5]

var doubled = numbers.map(function(n) {
  return n * 2
})
console.log(doubled) //=> [2,4,6,8,10]
```

[Source]

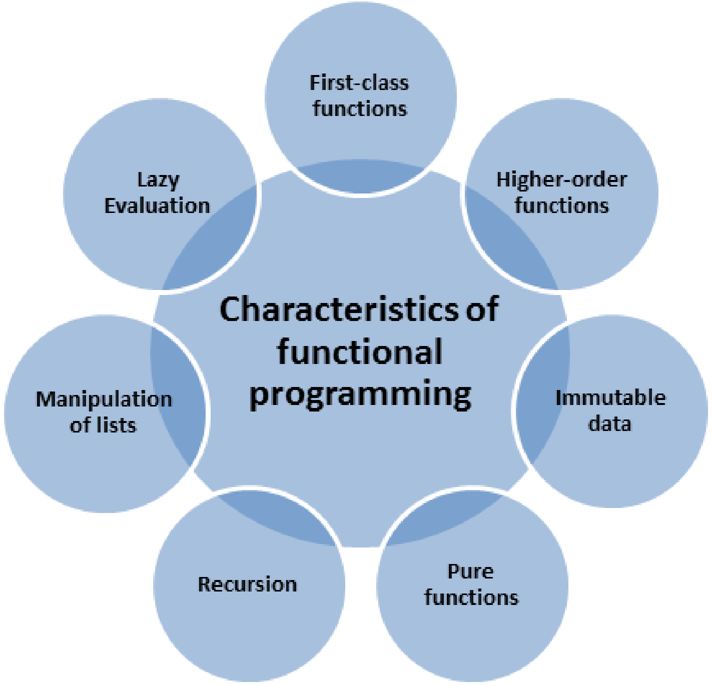
Software engineering process

Implementation – Programming Languages

Declarative Functional Programming

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);
var output = input.where( x -> x % 2 == 0);
```

- No sequence of steps
- A function (x % 2 == 0) is passed as a parameter to another function (where()) that is applied to an object (input)
- The evaluation does not change the internal program state
- Pure functional languages: Haskell, Hope, Mercury
- Java is not pure functional language



[Source]

Software engineering process

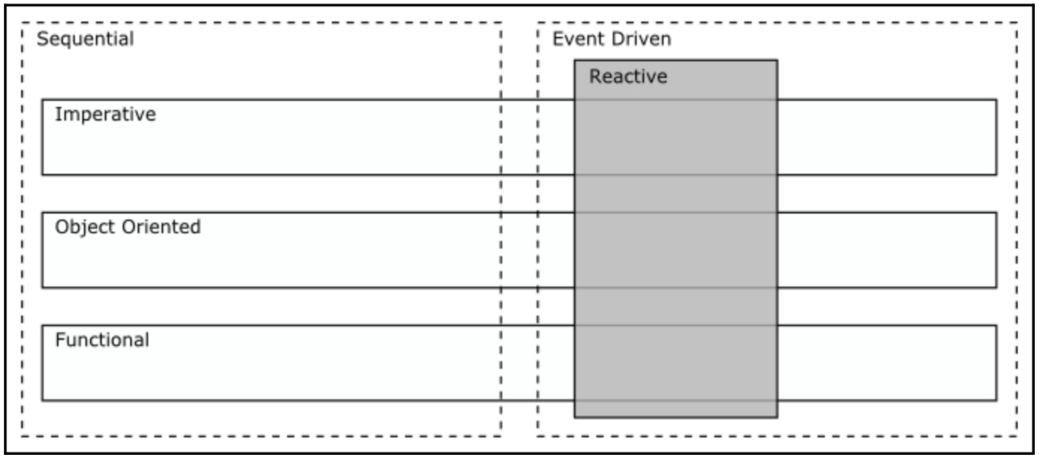
Implementation – Programming Languages

Reactive Programming

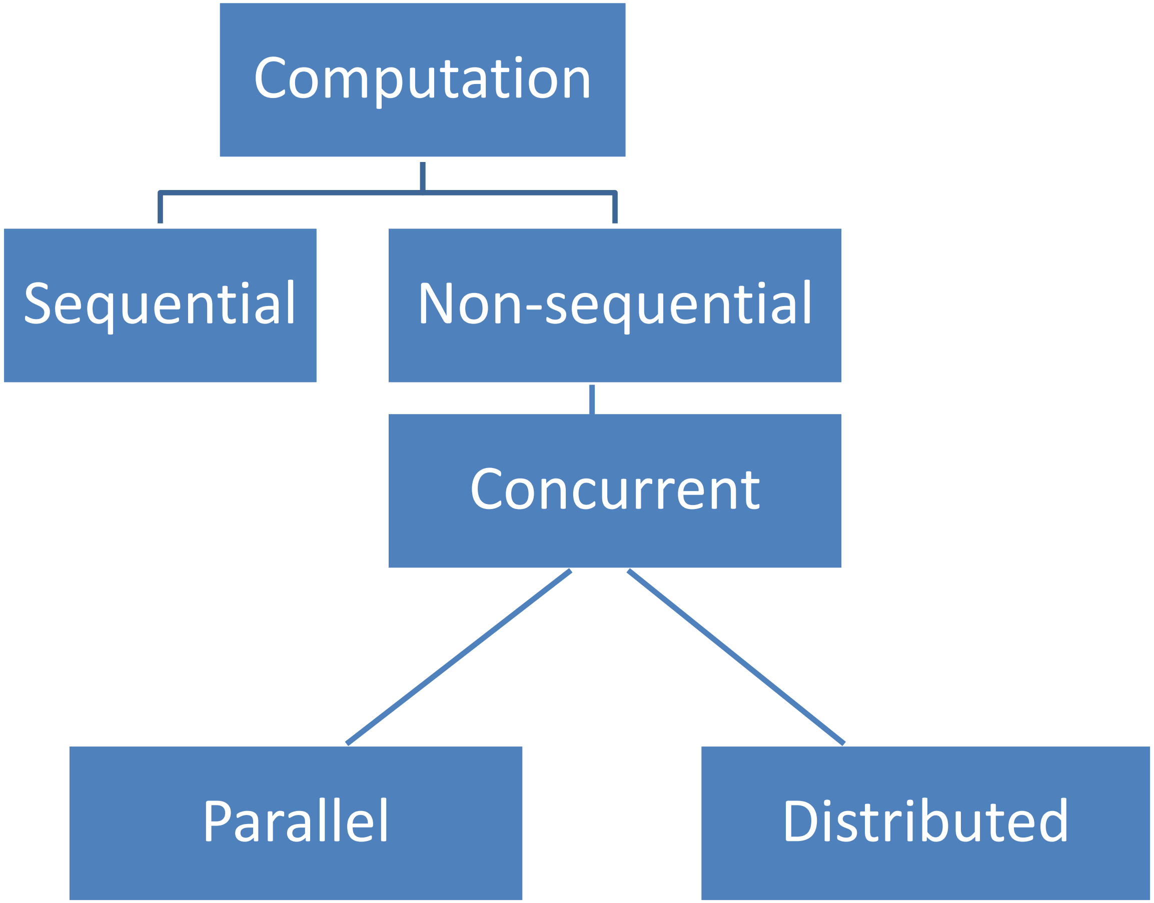
- Deals with data flows and propagation of data changes (data streams)
- Consider the assignment statement:

```
x = y + z;
```

  - If y and z changes values => x value will be modified accordingly
- Principles
  - Code “reacts” to events
  - Handles values as they vary in time, propagating changes to the code that uses those values
- Reactive programming – is not a programming paradigm of its own



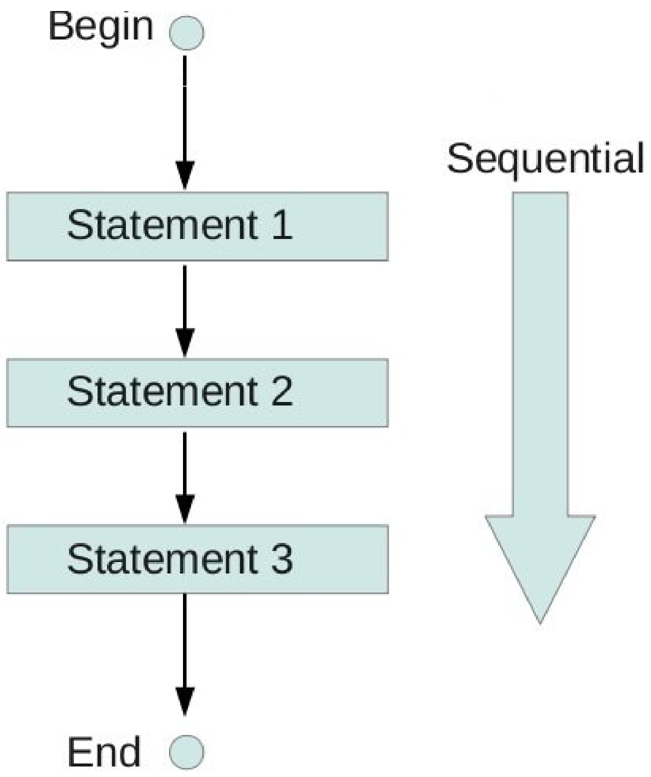
Software engineering process
Implementation – Types of Computation



Software engineering process

Implementation – Types of Computation

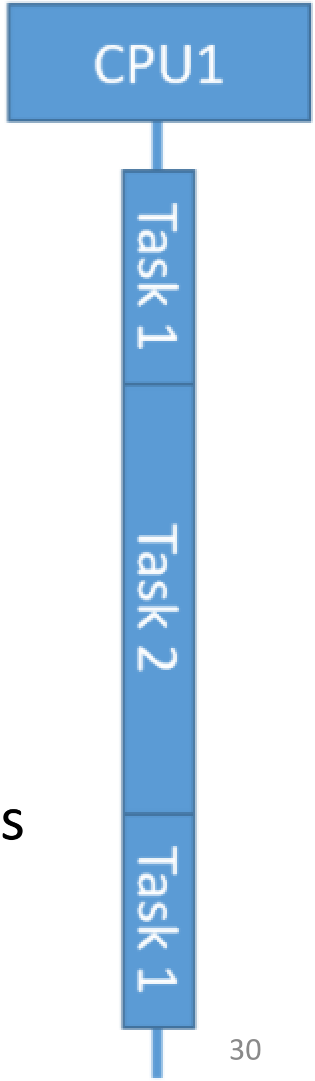
- **Sequential Programming**
  - Execution of one statement at a time
  - Usually executed on a single processor
    - Processor speed is important
  - Transforms input data to results according to the implemented algorithm
  - Usually, deterministic



Software engineering process

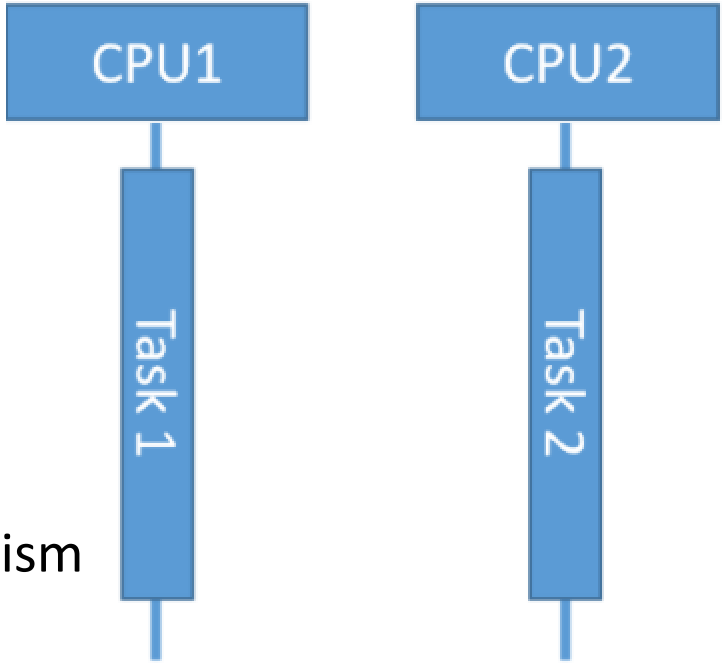
Implementation – Types of Computation

- **Concurrent Programming**
  - Concurrent program
    - Tasks executed simultaneously (in parallel)
  - Execution support
    - **Parallel** or **distributed** processes
  - Transform sequential to concurrent programming
    - Decompose large tasks into multiple smaller tasks
    - Assigning the smaller tasks to multiple workers to work on simultaneously
      - Mapping on computational resources
    - Coordinating the workers



Software engineering process
Implementation – Types of Computation

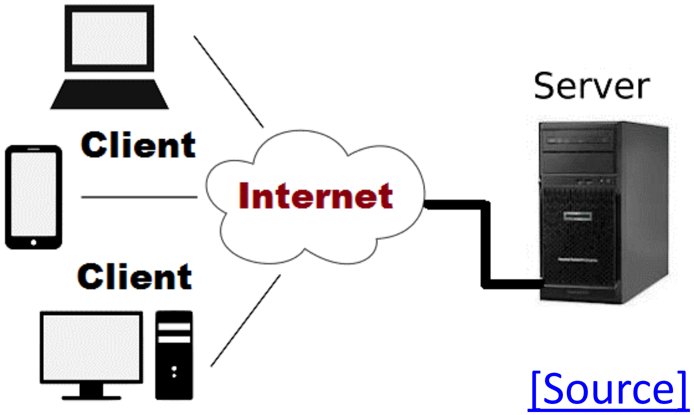
- **Parallel Programming**
  - Special form of concurrent programming
    - Multiple physical processors
  - Constructing parallel applications
    - Functional parallelism
    - Master–slave parallelism
    - Single Program Multiple Data parallelism
  - Parallel programming main tasks
    - Decomposing tasks
    - Distributing tasks
    - Coordinating tasks



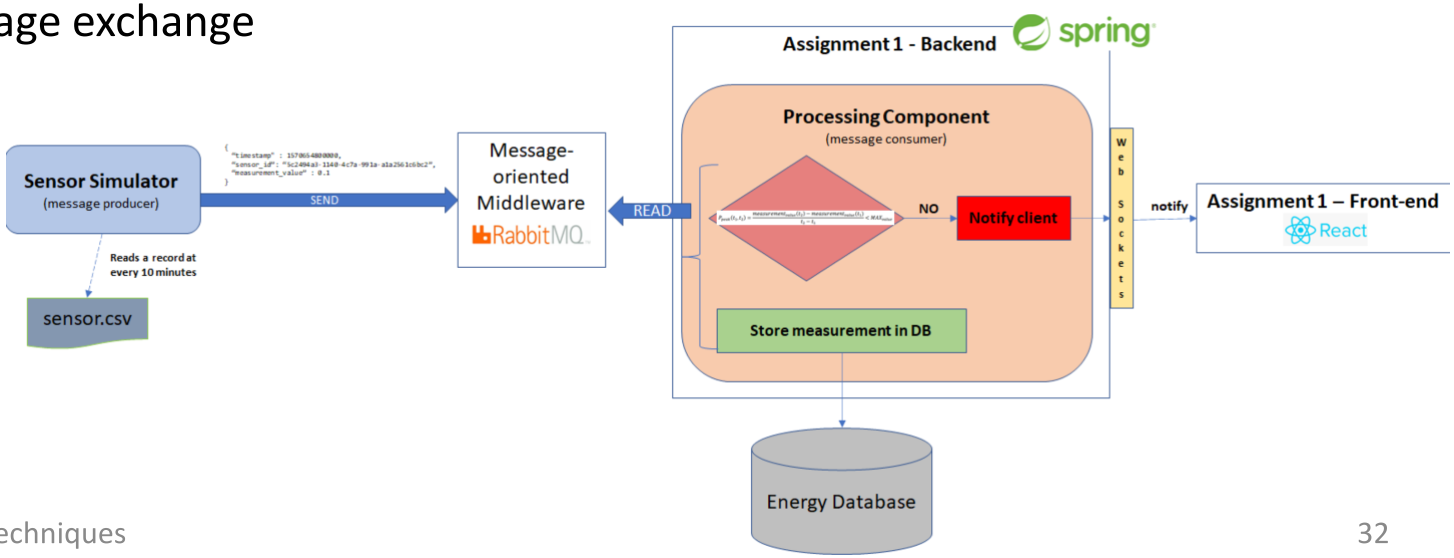
Software engineering process

Implementation – Types of Computation

- **Distributed Programming**
  - Special form of concurrent programming
    - multiple physical processors
    - remotely located
    - no shared memory
  - Inter-process communication
    - communication channels
    - message exchange



[Source]



```
{
  "timestamp" : 1570654800000,
  "sensor_id": "5c2494a3-1140-4c7a-991a-a1a2561c6bc2",
  "measurement_value" : 0.1
}
```

Software engineering process

Implementation – Types of Computation

- **Transformational (pipeline) Programming**
  - Data driven
  - Based on the sequence: read data -> process data -> output results
  - Can be seen as functions of n inputs generating m outputs
  - Program behavior depends on program current state and input
  - Can be parallelized

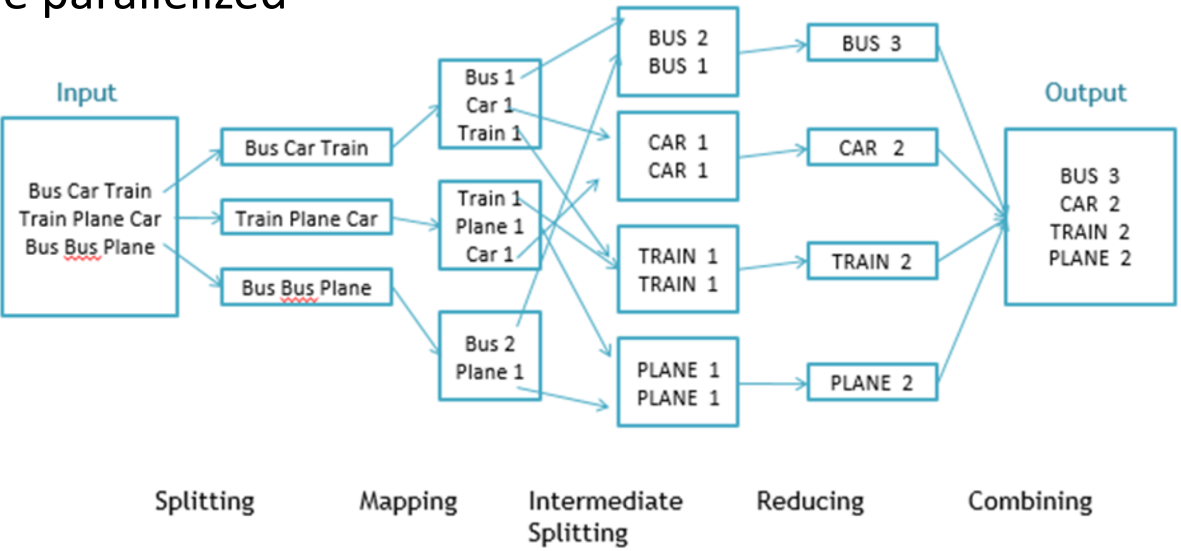
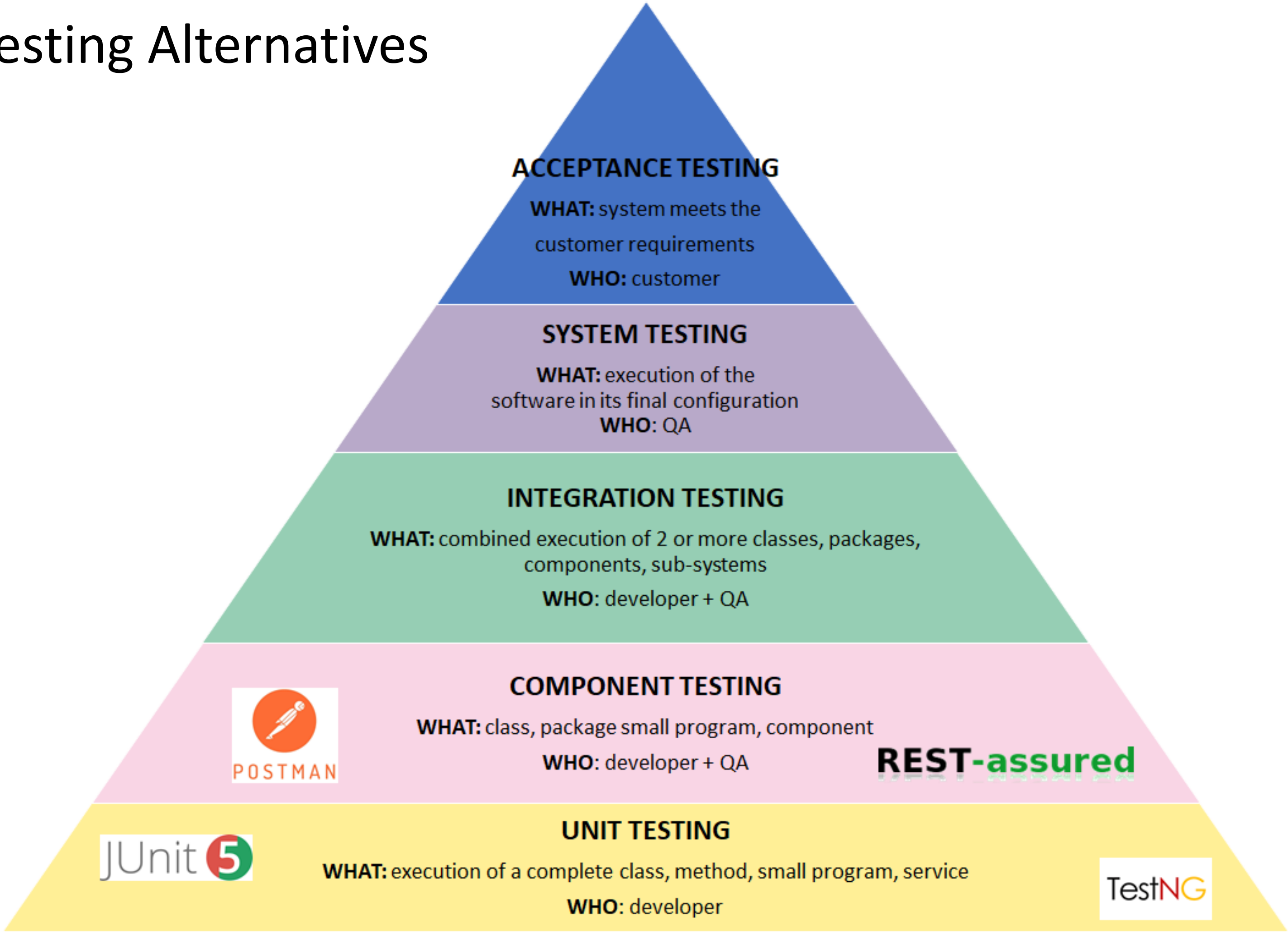


Fig. WorkFlow of MapReducing

[Source]

Software engineering process
Testing and Debugging

- Testing Alternatives



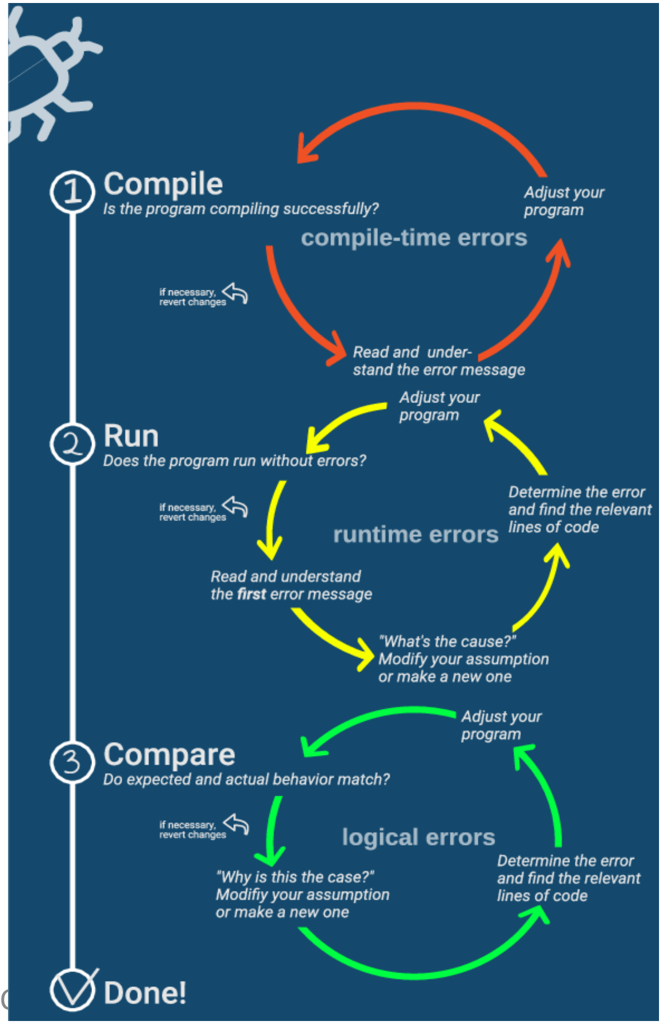
ACCEPTANCE TESTING
WHAT: system meets the customer requirements
WHO: customer
SYSTEM TESTING
WHAT: execution of the software in its final configuration
WHO: QA
INTEGRATION TESTING
WHAT: combined execution of 2 or more classes, packages, components, sub-systems
WHO: developer + QA
COMPONENT TESTING
WHAT: class, package small program, component
WHO: developer + QA
POSTMAN
REST-assured
UNIT TESTING
WHAT: execution of a complete class, method, small program, service
WHO: developer
JUnit 5
TestNG

[Source]

Software engineering process

Testing and Debugging

Debugging



① Compile
Is the program compiling successfully?
compile-time errors
Read and understand the error message
Adjust your program
if necessary, revert changes

② Run
Does the program run without errors?
runtime errors
Read and understand the **first** error message
"What's the cause?" Modify your assumption or make a new one
Determine the error and find the relevant lines of code
Adjust your program
if necessary, revert changes

③ Compare
Do expected and actual behavior match?
logical errors
"Why is this the case?" Modifiy your assumption or make a new one
Determine the error and find the relevant lines of code
Adjust your program
if necessary, revert changes

Done!

- Type of errors
- Techniques
  - Pay attention to the error message
  - Use debuggers
  - Add debug messages or comment part of code
  - Check previous working versions
  - Rubber duck
  - Reproduce the problem
  - **Take a break**

[Source]