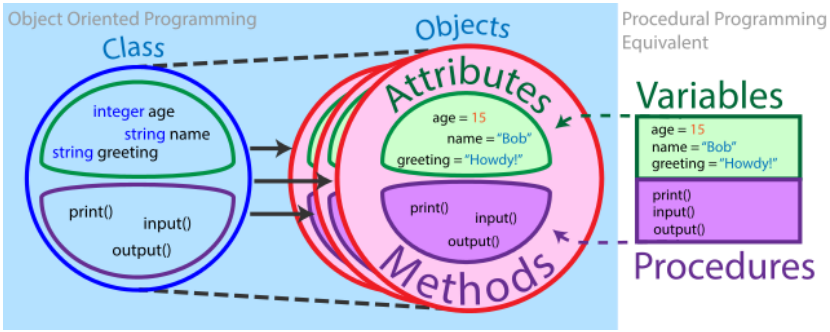
Object Oriented Software Principles and Design



Computer Science CS 611
Boston University

Christine Papadakis-Kanaris

What is Computer Science?

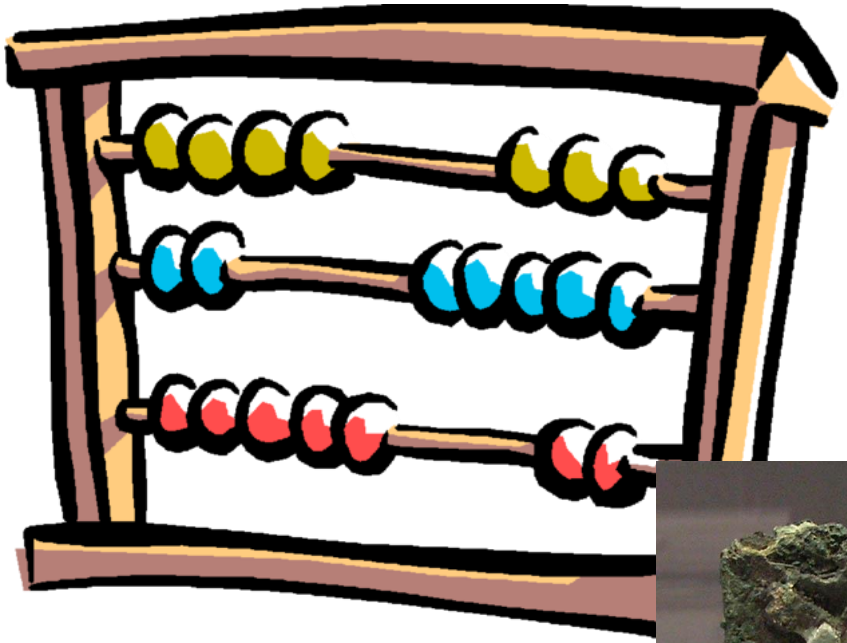
Computer Science is an embodiment of human intelligence and ingenuity. It is the byproduct of our quest to understand the universe, our need to find solutions to known problems, and a desire to free us from our ***own physical limitations****.*

Milestones in Computer Science

If I were to ask you to name one of the first important developments on the road to the modern day computer, what would you say?

Inventions along the way…

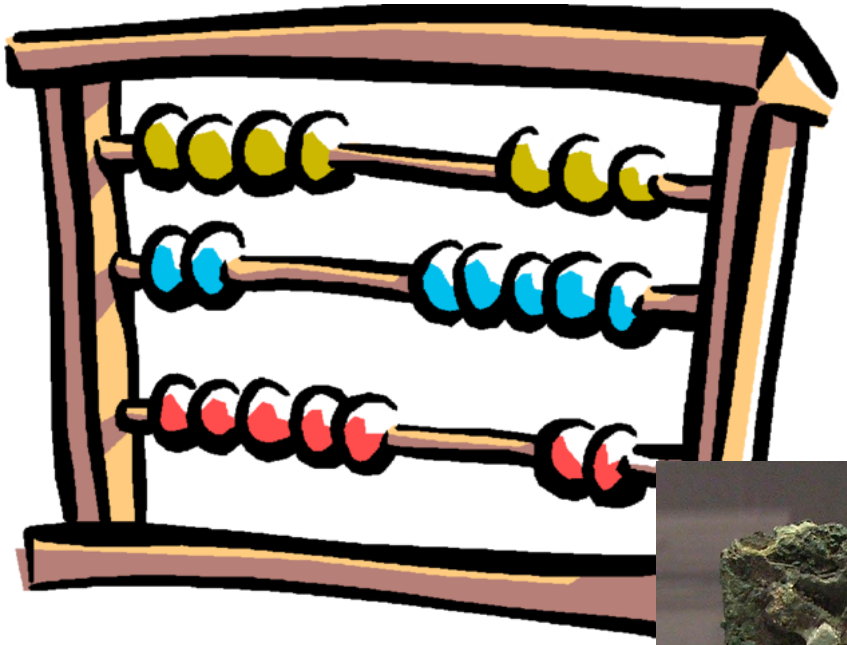
Antikythera Machine (2nd-1st century B.C.)



The Antikythera Wreck

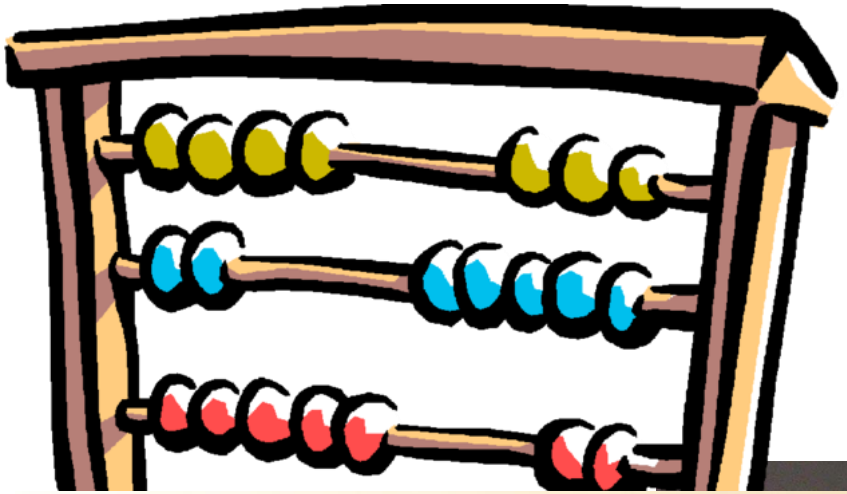
Inventions along the way…

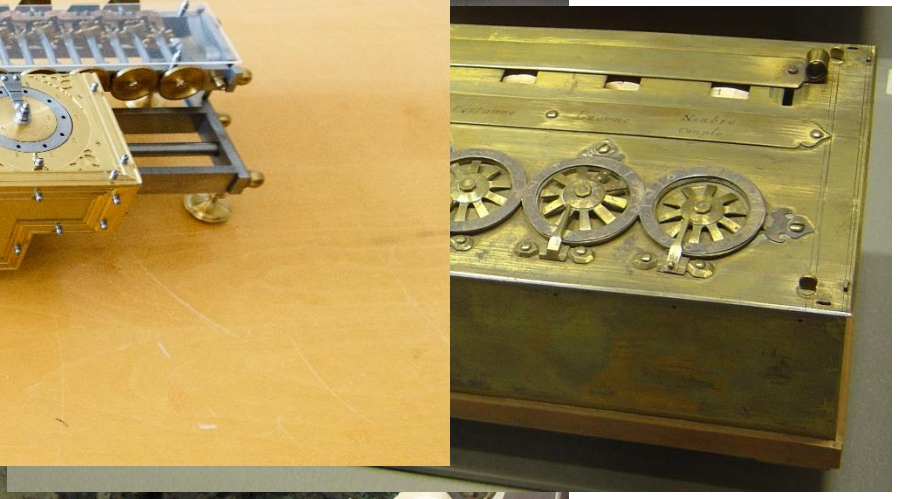
Pascal's Calculator (mid 17th century)



Inventions along the way…

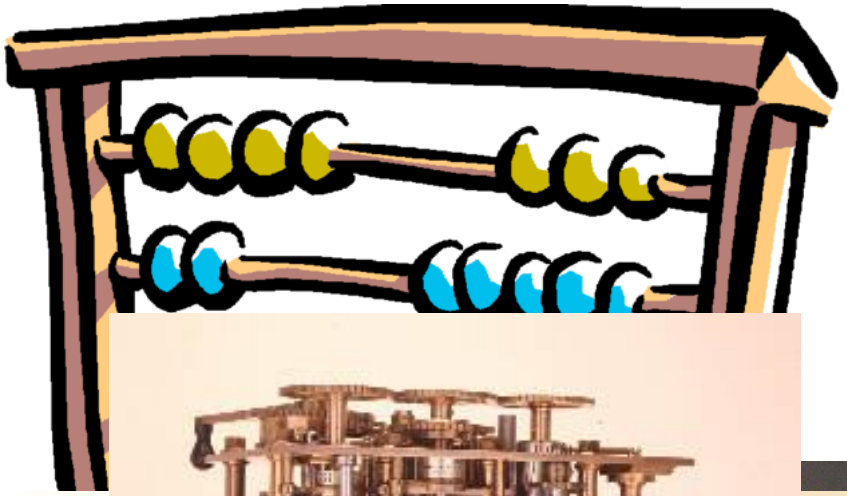
Leibniz wheel *(mid 17th century)*

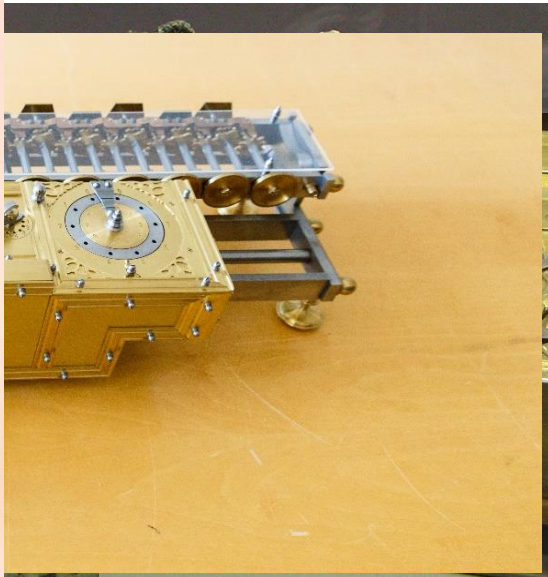




Inventions along the way…

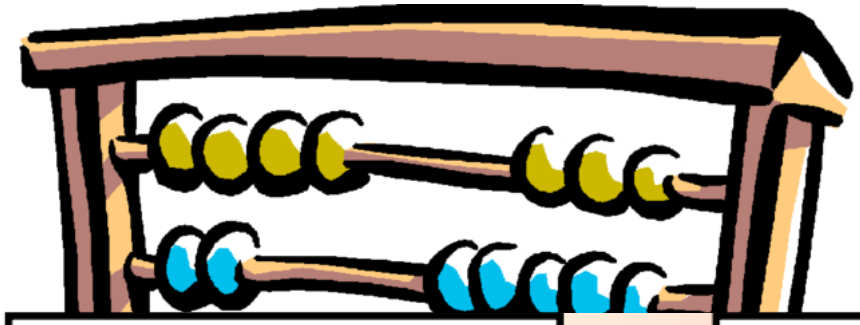
Difference Engine *(early 19th century)*

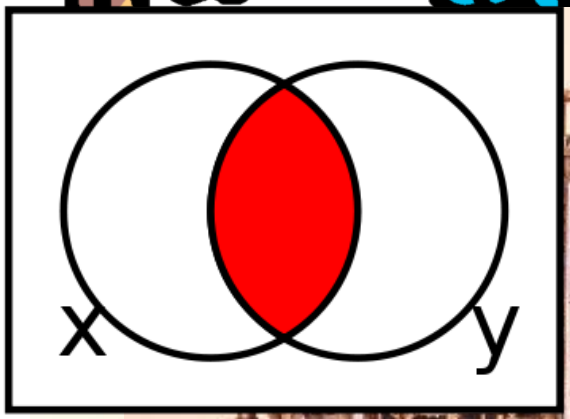


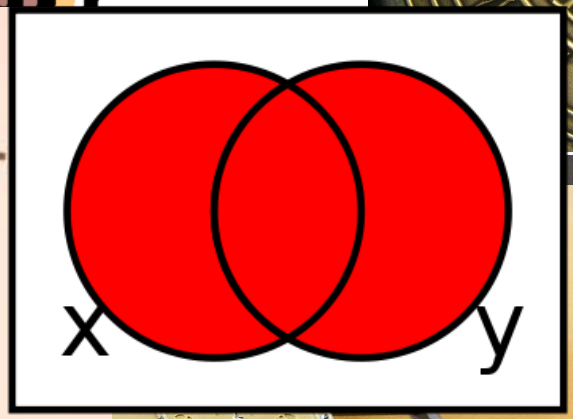


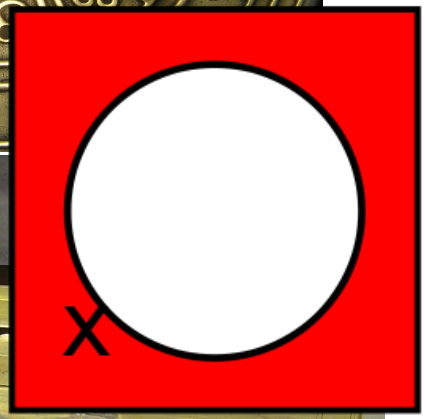
Inventions along the way…

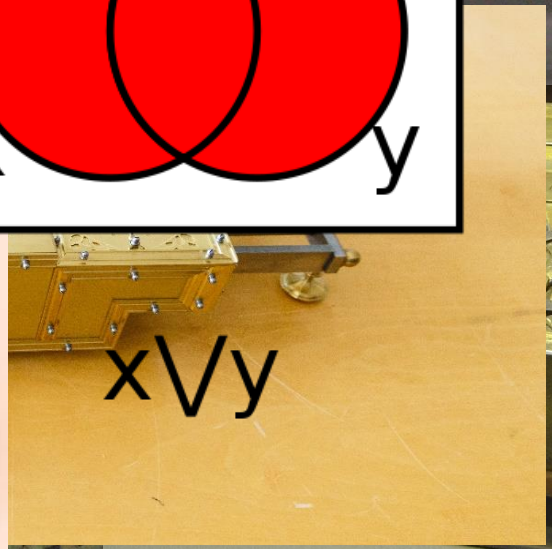
Boolean Logic

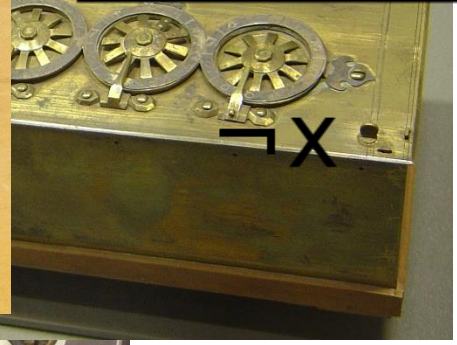










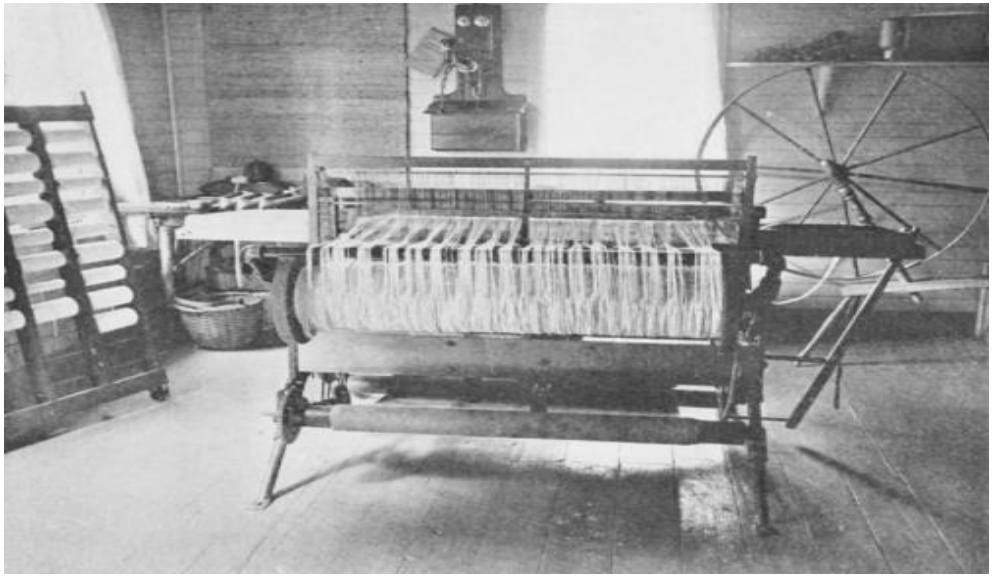


Milestones in Computer Science

If I were to ask you to name one of the first important developments on the road to the modern day computer, what would you say?

How many of you would believe that it came out of the textile industry?

Well, it did!!!



Milestones in Computer Science

1801: In France, Joseph Marie Jacquard invents a loom that uses **punched wooden cards** to automatically weave fabric designs.

Jacquard Loom in action

Why is this significant?

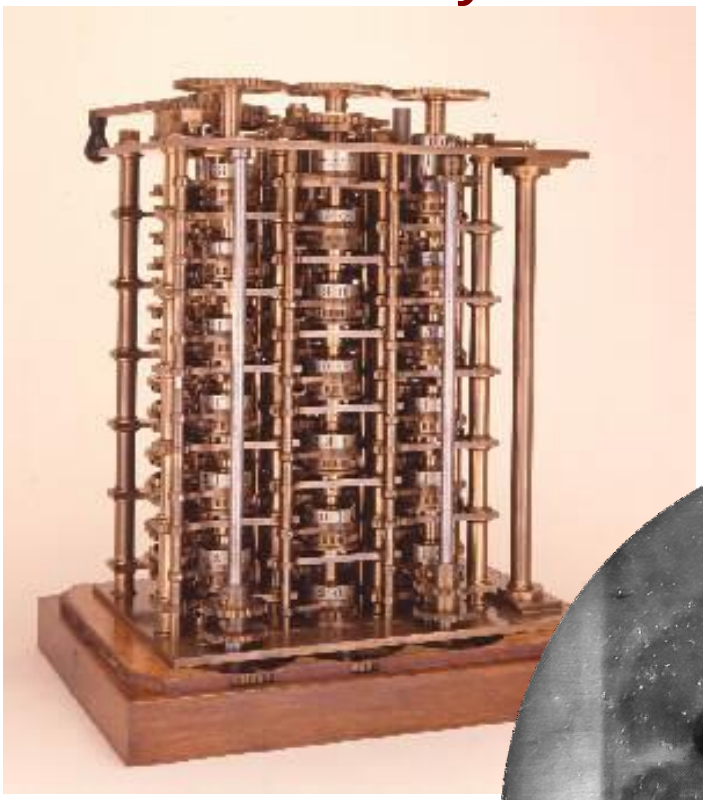
Milestones in Computer Science

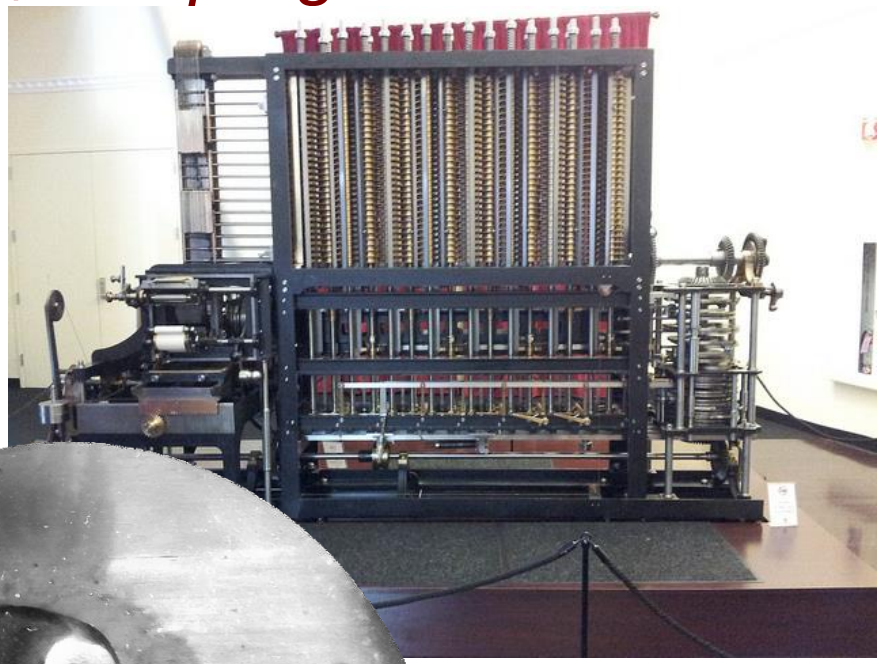
1801: In France, Joseph Marie Jacquard invents a loom that uses **punched wooden cards** to automatically weave fabric designs.

Why is this significant?

In doing so, Joseph Marie Jacquard invents the first "programmable" device.

Augusta Ada Lovelace
Lady Lovelace, *first programmer*





Lady Lovelace's notes on the Analytical Engine

Milestones in Computer Science

1801: In France, Joseph Marie Jacquard invents a loom that uses **punched wooden cards** to automatically weave fabric designs.

Why is this significant?

In doing so, Joseph Marie Jacquard invents the first "programmable" device.

Who would believe that another major milestone came from a statistician working for the U.S. census bureau?

Well, it did!

1890: Herman Hollerith designed and built programmable card processing machines used to calculate the 1890 census, accomplishing the task in just two years and saving the government millions!

Hollerith Tabulator

And the future (as we know it) is established!

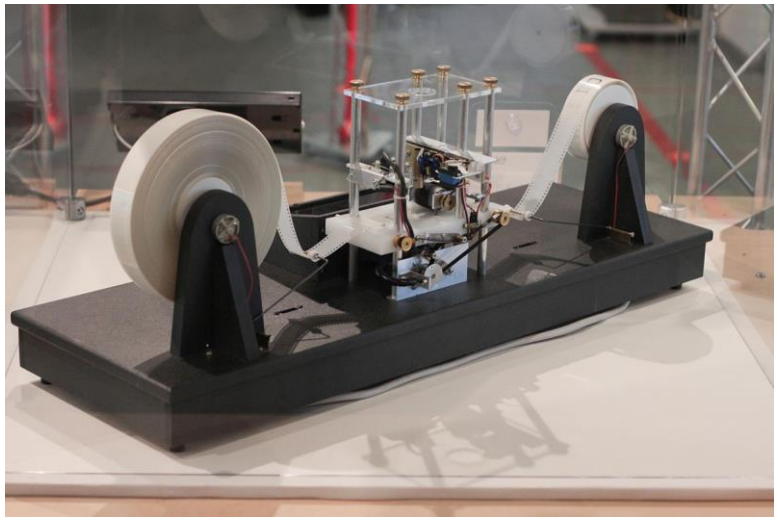
You have to believe that anyone who completes a task years ahead of estimates **saving $$$$** is going to get noticed. And he was!

Together with several investors, Herman Hollerith establishes a little company which ultimately became known as….

IBM!!!

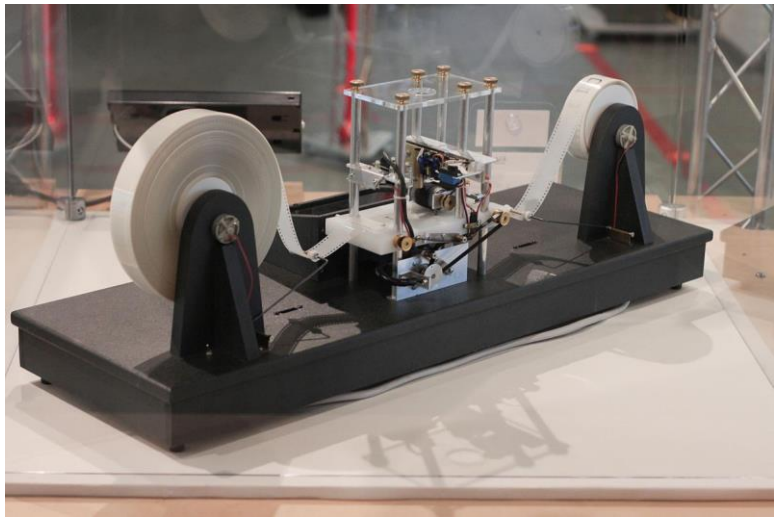
The exponential growth of the 20th century:

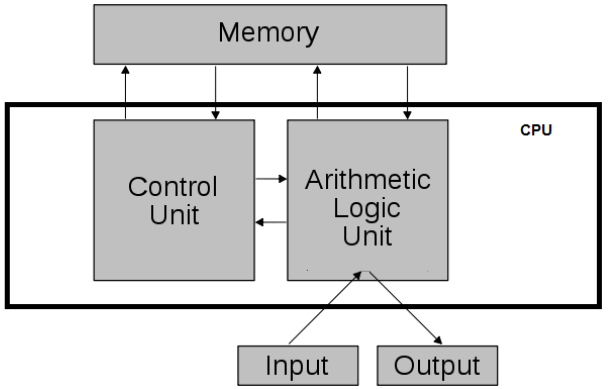
Turing Machine



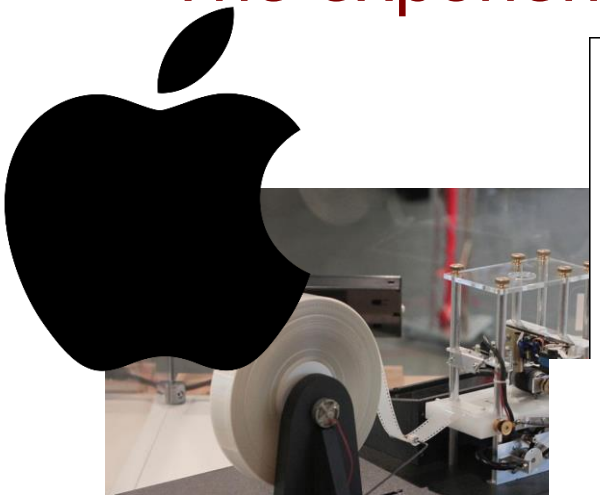
The exponential growth of the 20th century:

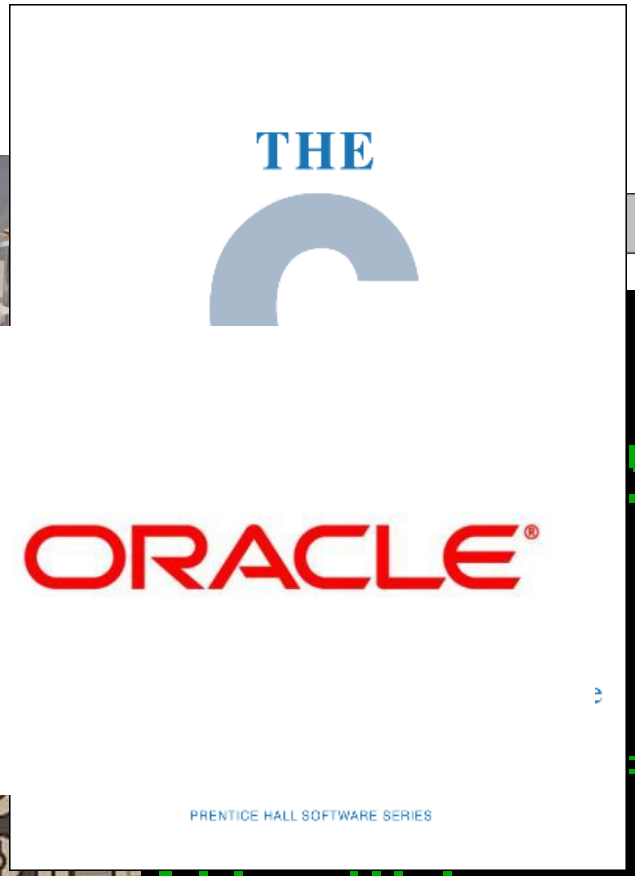
Von Newman Architecture





The exponential growth of the 20th century





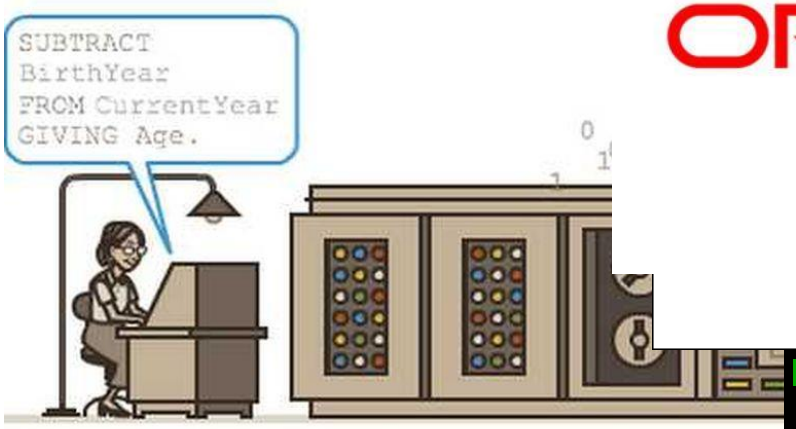
Memory

uter Products
79,80,81 by Se

= 0

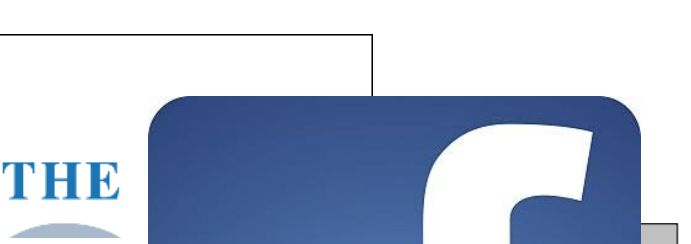
A:hexZbin mon

A:_



Google Search I'm Feeling Lucky

The exponential growth of the 20th century

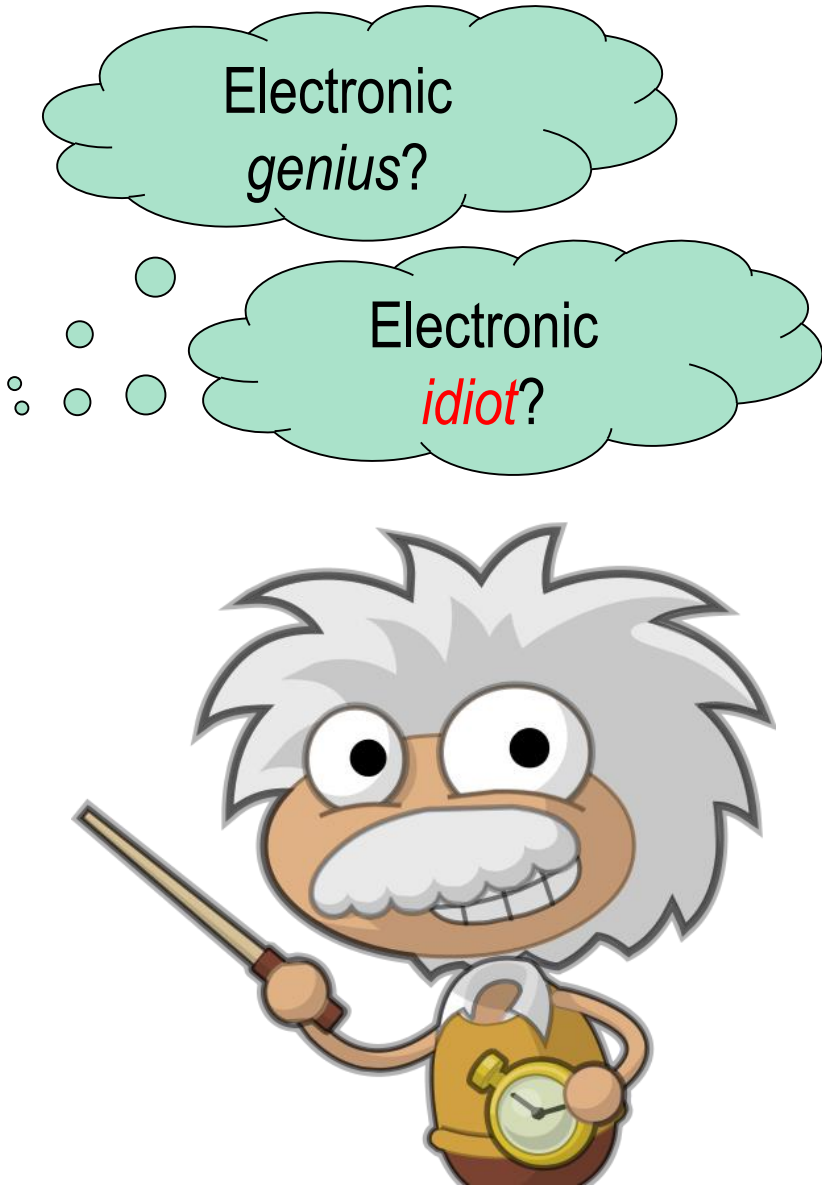


Google Search I'm Feeling Lucky

[History on Computing]

Fundamental Question in Computer Science

- Are computers intelligent?



Fundamental Question in Computer Science

- Are computers intelligent?
- A computer is just a **black box**

A (binary) device that responds to two types of signals

on and off!

- *A symbol processing machine which does exactly what we program it to do – nothing more and nothing less.*

Fundamental Question in Computer Science

- Are computers intelligent?
- A computer is just a **black box**

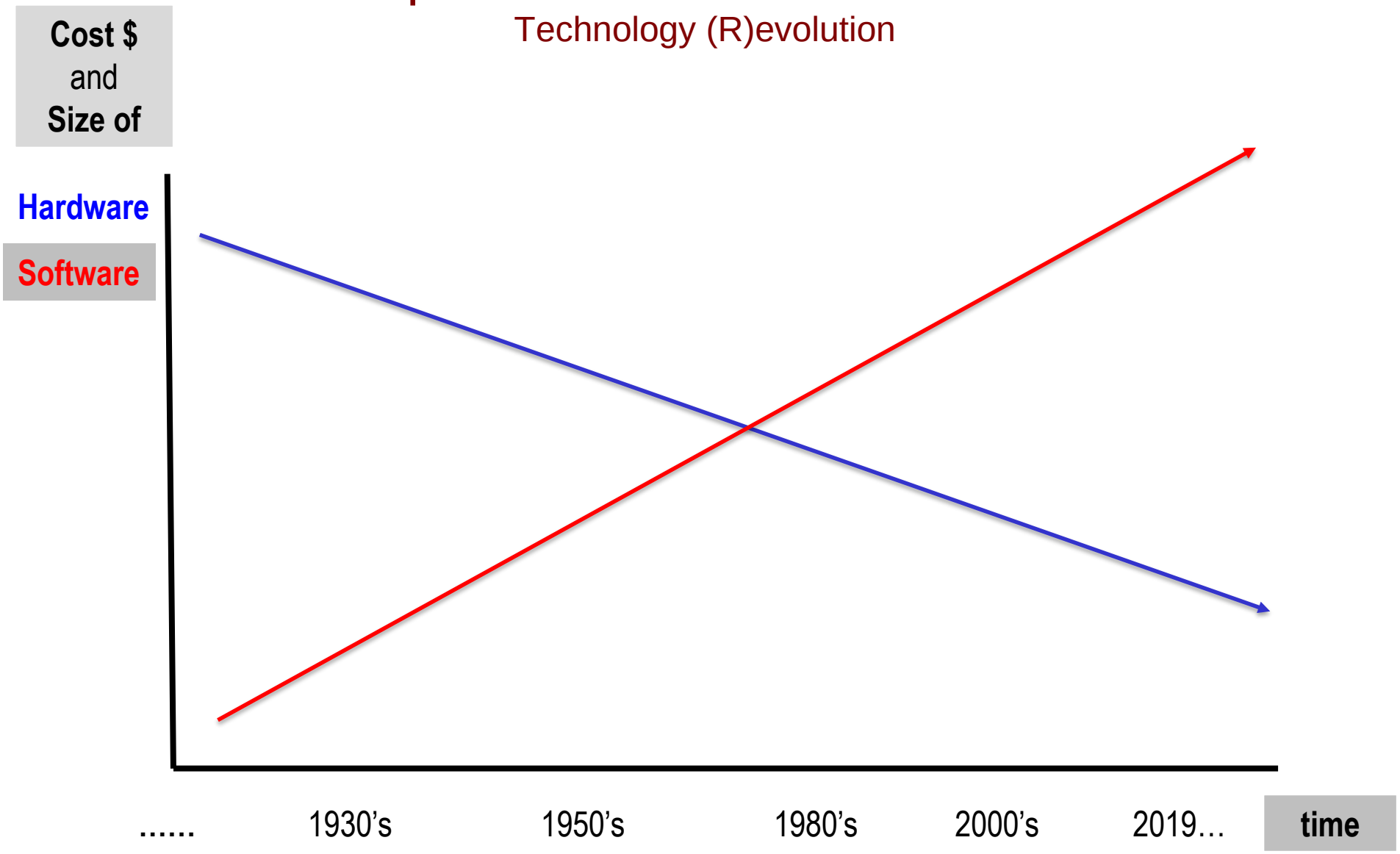
A (binary) device that responds to two types of signals

on and off!

*It is the layers of **software** executing on a computer that make computers interesting and give the illusion of intelligence!*

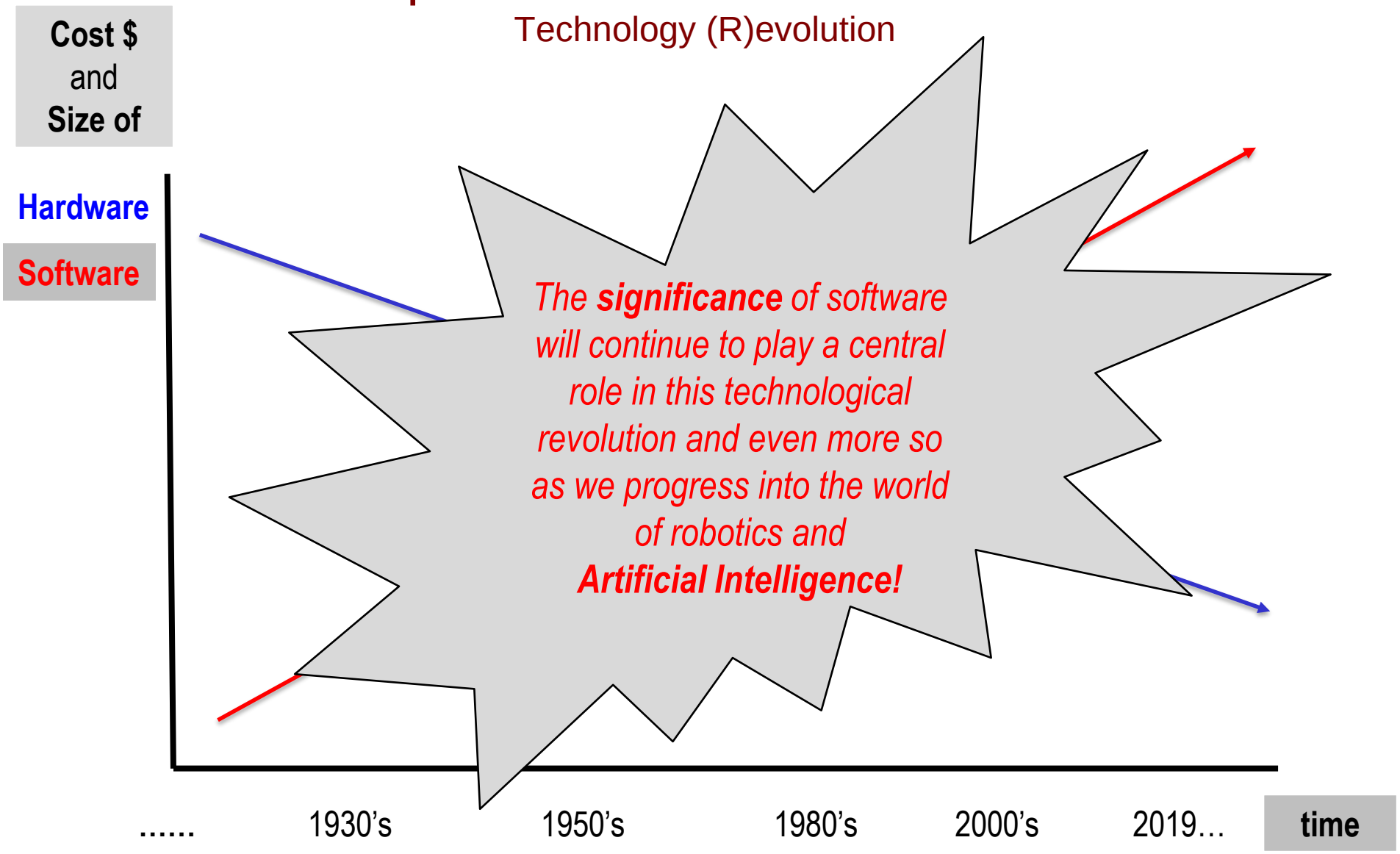
Importance of Software in the

Technology (R)evolution



Importance of Software in the

Technology (R)evolution



Software and Intelligence:
Human Intelligence

- Senses
- Memory
  - Short term
  - Long term

> *"You can mass-produce hardware; you cannot mass-produce software; you cannot mass-produce the human mind."*
>
> *Michio Kaku*

- Reason or Logic
  - We use reason and logic to make decisions
- Repetition
  - Once we learn to do something, we can do it over and over!
- Optimization
  - Once we learn how to do something, we try to do it better!

Software and Intelligence:
Program Intelligence

- Senses ⇨ Input/Ouput (I/O)
- Memory
  - Short term ⇨ Data Types and Variables, **and Data structures**
  - Long term ⇨ Files
- Reason or Logic ⇨ Conditional Statements
  - We use reason and logic to make decisions
- Repetition ⇨ Loops
  - Once we learn to do something, we can do it over and over!
- Optimization ⇨ Functions, OO Programming
  - Once we learn how to do something, we try to do it better!

Software and Intelligence:
Program Intelligence

- Senses ⇨ Input/Ouput (I/O)
- Memory
  - Short term ⇨ Data Types and Variables, **and Data structures**
  - Long term ⇨ Files
- Reason or Logic ⇨ Conditional Statements
  - We use reason and logic to make decisions
- Repetition ⇨ Loops
  - Once we learn to do something, we can do it over and over!
- **Optimization** ⇨ Functions, **OO Programming**
  - Once we learn how to do something, we try to do it better!

Motivation for Innovation

We have a problem to solve…

Understand the problem

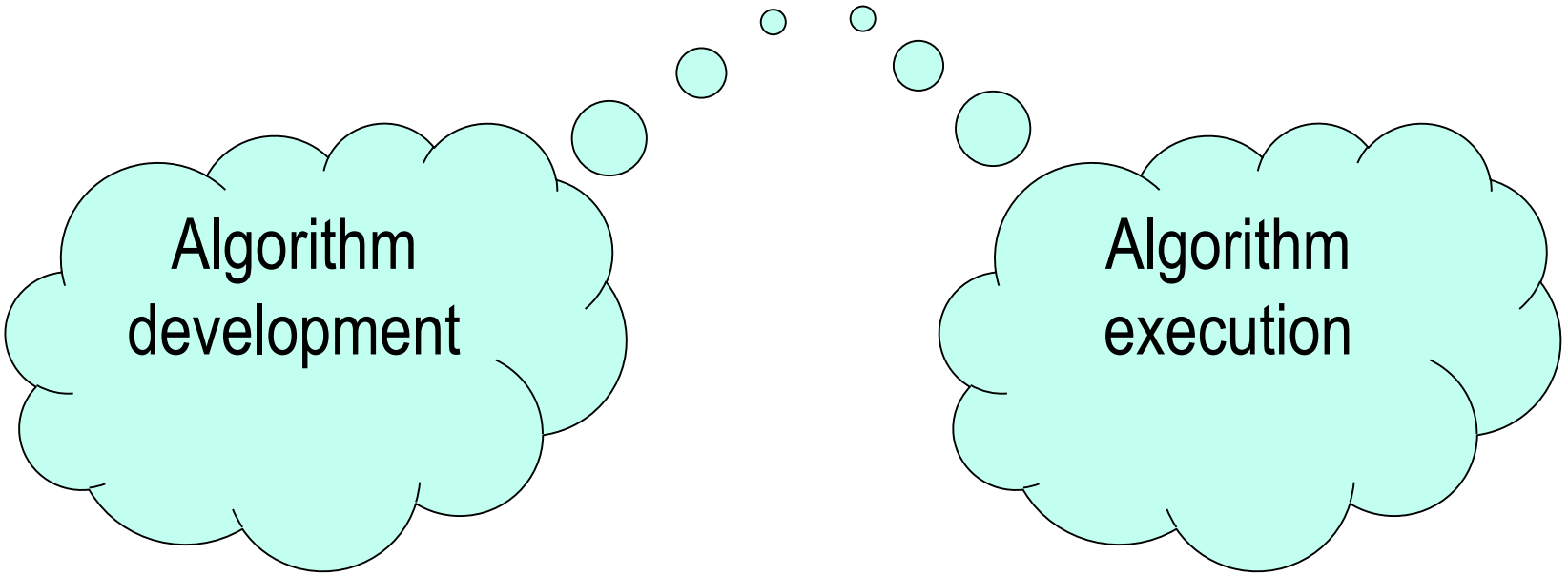
Identify the solution

Express the solution

Algorithm

Algorithm

A well ordered collection of **unambiguous** and **effectively computable** operations that when executed produces a **result** and **halts** in a finite amount of time



What do each of these *advancements* or *milestones* have in common?

Jacquard's loom
Analytical Engine
Hollerith's Machines
The Turing Machine
.
.
The Algorithm

*They solved the problem by creating an **abstract** model of the problem!*

Abstraction

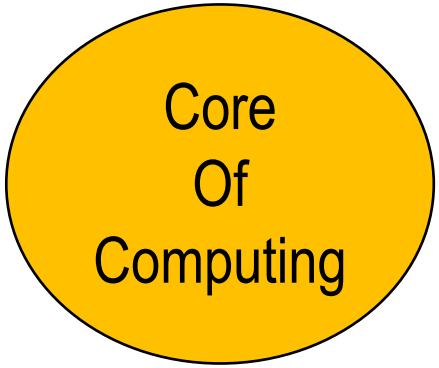
"The essence of abstraction is preserving information that is relevant in a given context and **forgetting** information that is irrelevant in that context"

John V Guttag

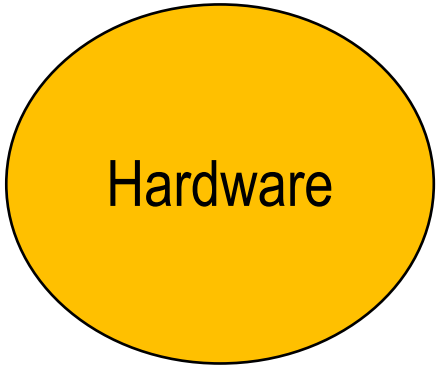
"Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. --

Edsger Dijkstra

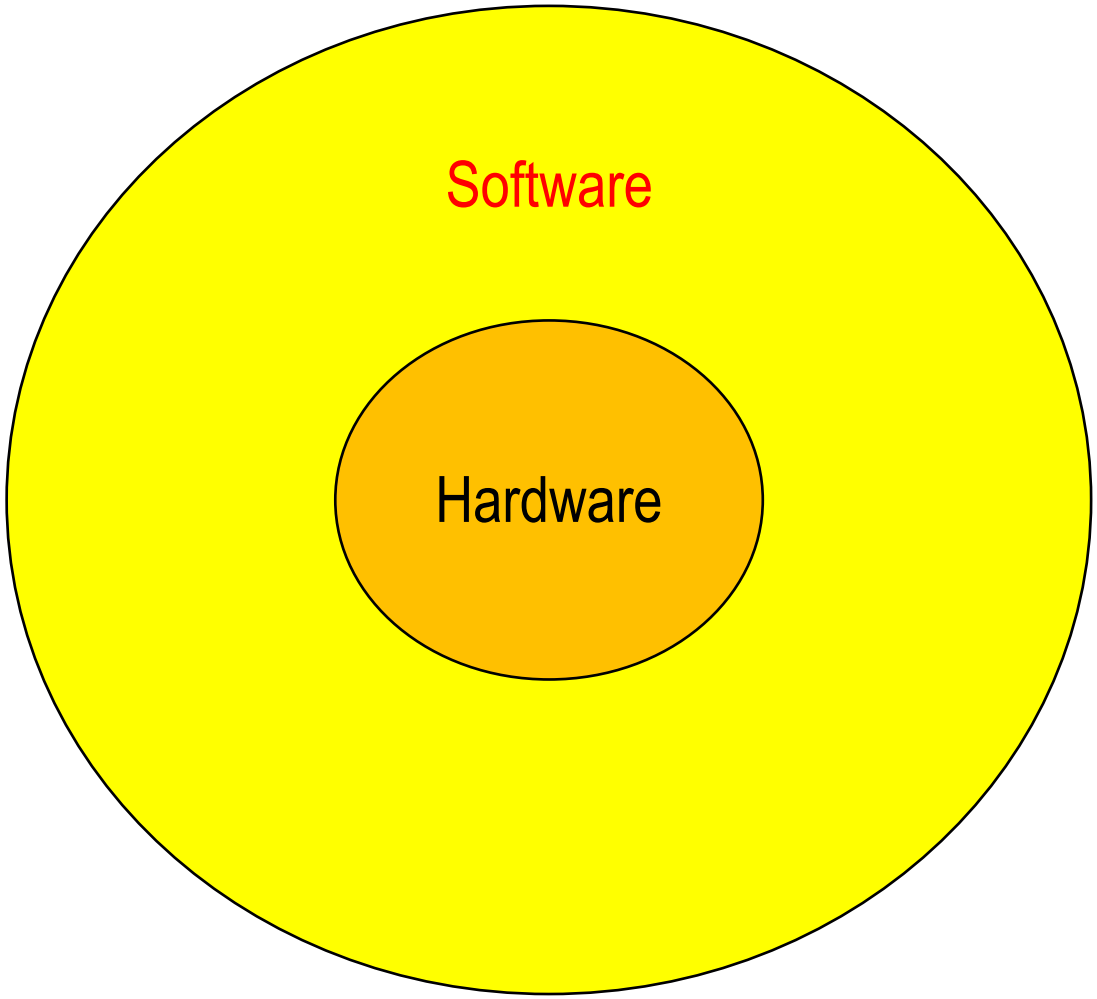
Power of Abstraction



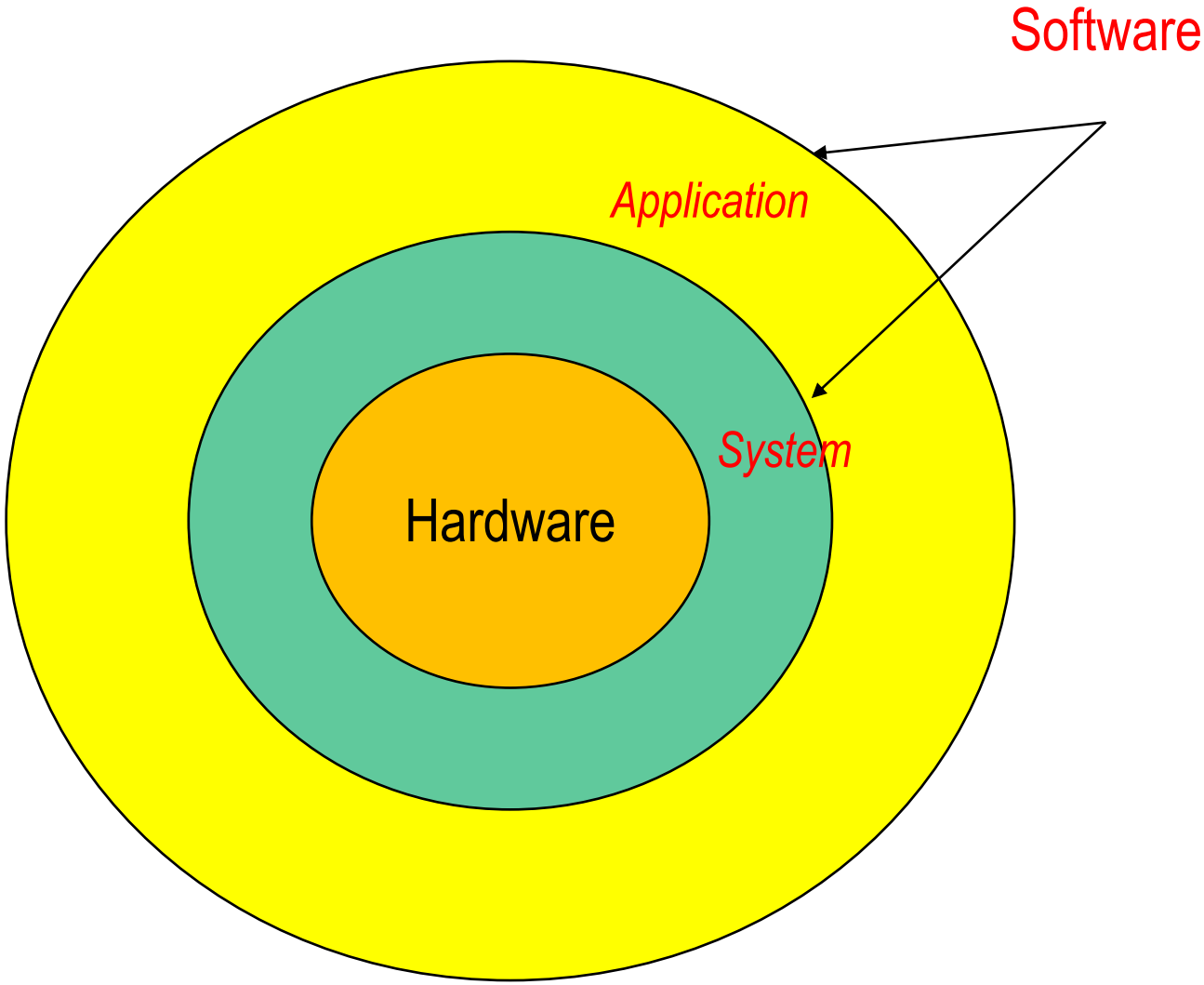
Power of Abstraction



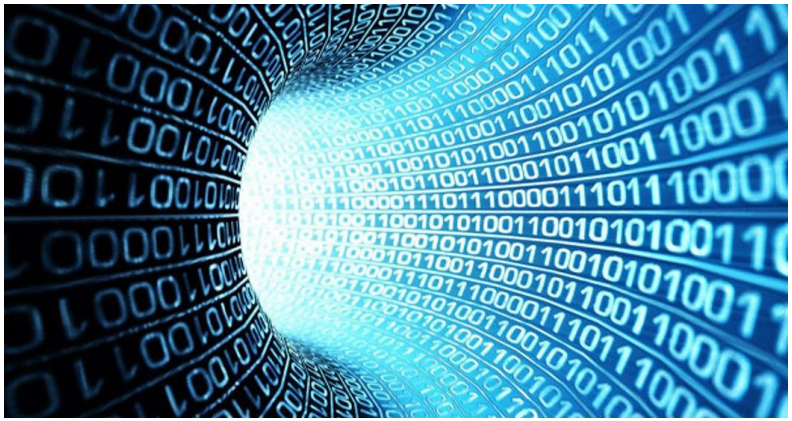
Power of Abstraction



Power of Abstraction



Abstraction in Software…



Machine Language

The only language a computer understands!

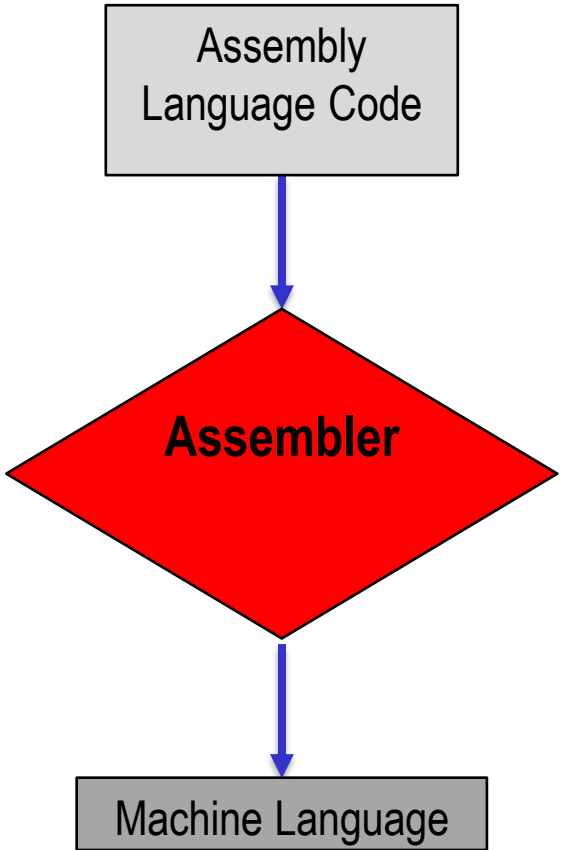
- Lowest level representation of an instruction set that a given computer can execute.
- Instructions must be executed directly by a computer's CPU
- Machine or Architecture dependent. You need to know the architecture, program registers, individual bit patterns, etc

Abstraction in Software…

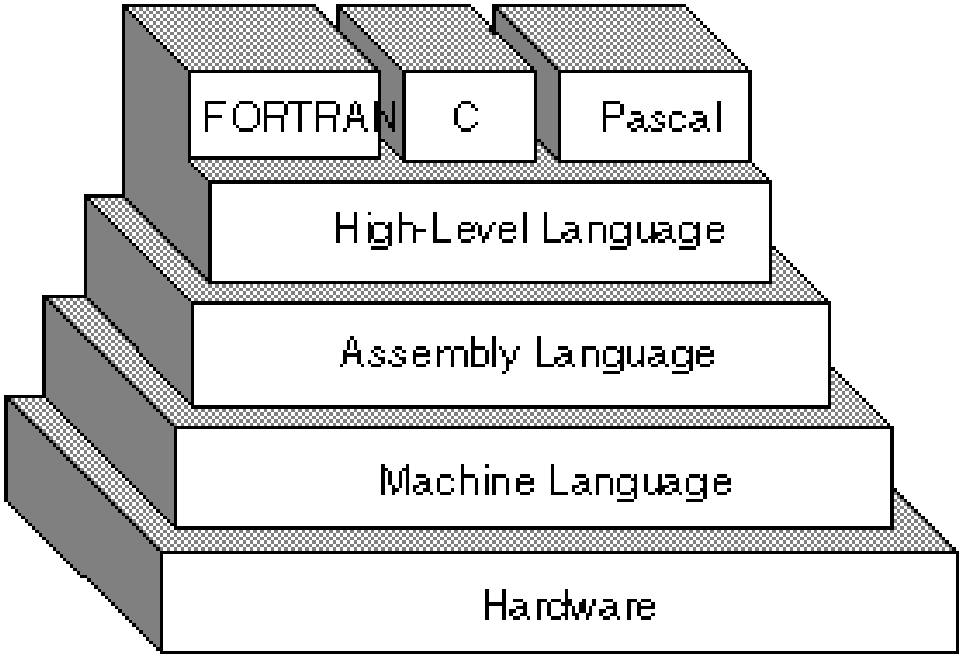
```
Line No.
 1              .data
 2              .align 4
 3         A:   .long  6, -2, 3, 7, -1
 4              .comm  min,4,4
 5              .comm  max,4,4
 6              .section .rodata
 7         fmt: .string "min = %d, max = %d\n"
 8              .text
 9              .globl  main
10         main:                          # instruction size
11              pushl  %ebp               # 1 byte
12              movl   %esp, %ebp         # 2 bytes
13              movl   A, %eax            # 5 bytes
14              movl   %eax, min          # 5 bytes
15              movl   %eax, max          # 5 bytes
16              movl   $1, %ecx           # 5 bytes
17         loop:
18              cmpl   $5, %ecx           # 3 bytes
19              jge    done               # 2 bytes
20              movl   A(,%ecx,4),%eax    # 7 bytes
21              cmpl   min, %eax          # 6 bytes
22              jge    next1              # 2 bytes
23              movl   %eax, min          # 5 bytes
24         next1:
25              cmpl   max, %eax          # 6 bytes
26              jle    next2              # 2 bytes
27              movl   %eax, max          # 5 bytes
28         next2:
29              incl   %ecx               # 1 byte
30              jmp    loop               # 2 bytes
31         done:
32              pushl  max                # 6 bytes
33              pushl  min                # 6 bytes
34              pushl  $fmt               # 5 bytes
35              call   printf             # 5 bytes
36              addl   $12, %esp          # 3 bytes
37              movl   $0, %eax           # 5 bytes
38              leave                     # 1 byte
39              ret                       # 1 byte
```

Assembly Language

A higher level *machine* language!

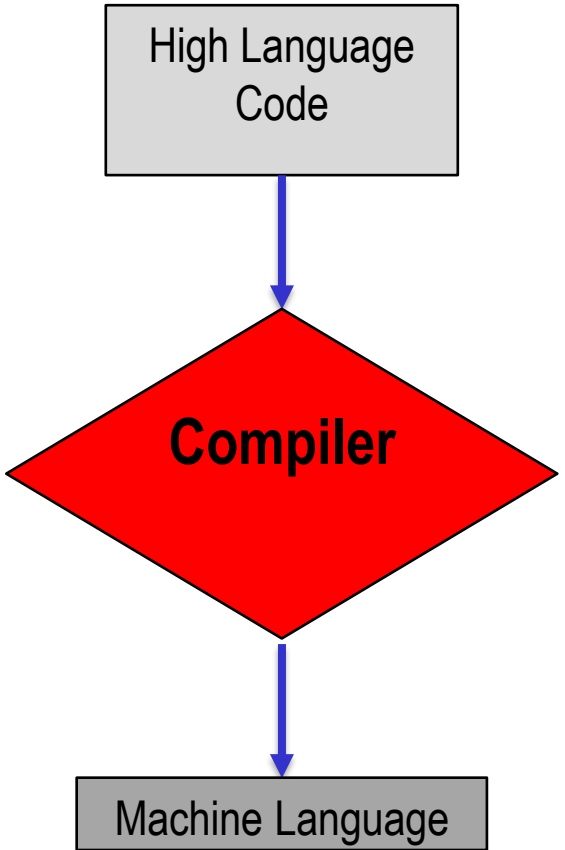


Abstraction in Software…

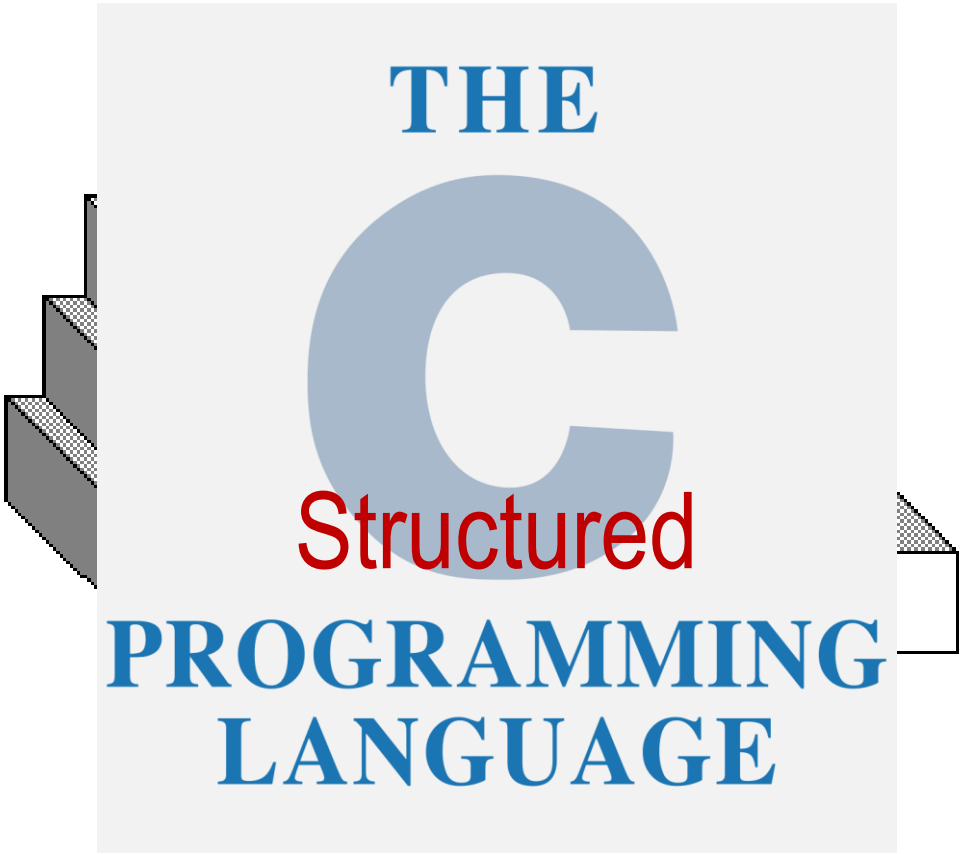


High Level Language

A higher level *assembly* language!



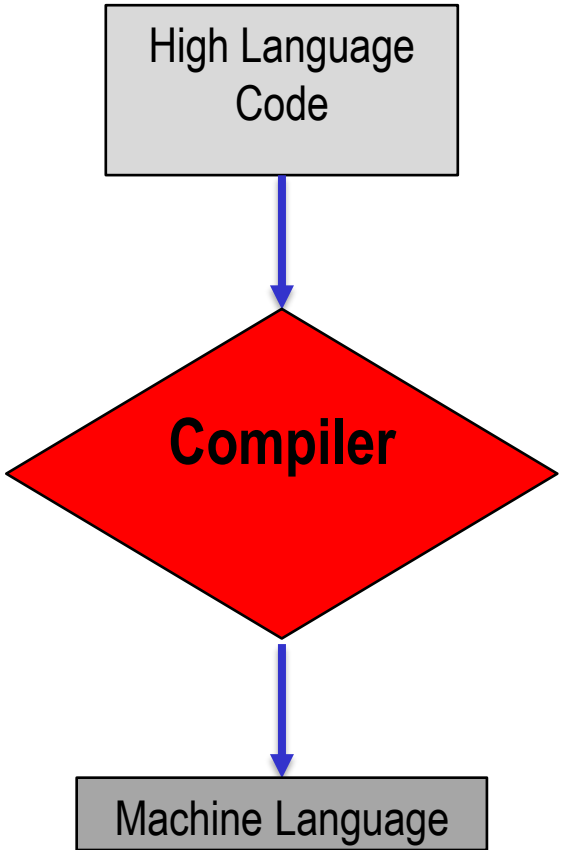
Abstraction in Software…



Modularize programming … more specifically, allowed us to write programs in logical modules…
logical building blocks.

High Level Language

A higher level language!



Dijkstra's model

A program is broken down into logical "sub" modules that each have **one point of entry** and **one point of exit**…

- We can treat each "sub" module as an independent entity that can be called or invoked by another programs.
- Facilitates a **modular** approach to designing programs.
- "Sub" program become building blocks to building large scale systems.

Limitations to Structured Programming

The ***data*** and the ***functions*** that process the data are **independent** of each other.

Object Oriented Programming:
an evolution of Structured Programming

*Couples the **logic** being performed with the data it is being performed on to create a physical entity that models a real life object.*

Object Oriented programming allows this coupling to be defined implicitly within the language!

In the world of OO the buildings blocks are no longer the code modules, but the physical entities or objects that are created!

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - ***A class should have only one reason to change.***

- Open Closed Principle
  - *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

- Liskov Substitution Principle
  - *Derived types must be completely substitutable for their base types.*

- Interface Segregation Principle
  - *Clients should not be forced to depend upon interfaces that they don't use.*

- Dependency Inversion/Injection
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - *A class should have only one reason to change.*

- Open Closed Principle
  - ***Software entities like classes, modules and functions should be open for extension but closed for modifications.***

- Liskov Substitution Principle
  - *Derived types must be completely substitutable for their base types.*

- Interface Segregation Principle
  - *Clients should not be forced to depend upon interfaces that they don't use.*

- Dependency Inversion/Injection
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - *A class should have only one reason to change.*

- Open Closed Principle
  - *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

- Liskov Substitution Principle
  - ***Derived types must be completely substitutable for their base types.***

- Interface Segregation Principle
  - *Clients should not be forced to depend upon interfaces that they don't use.*

- Dependency Inversion/Injection
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - *A class should have only one reason to change.*

- Open Closed Principle
  - *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

- Liskov Substitution Principle
  - *Derived types must be completely substitutable for their base types.*

- Interface Segregation Principle
  - ***Clients should not be forced to depend upon interfaces that they don't use.***

- Dependency Inversion/Injection
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

Coding vs. Programming
Understanding Design Principles

- Single Responsibility Principle
  - *A class should have only one reason to change.*

- Open Closed Principle
  - *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

- Liskov Substitution Principle
  - *Derived types must be completely substitutable for their base types.*

- Interface Segregation Principle
  - *Clients should not be forced to depend upon interfaces that they don't use.*

- Dependency Inversion/Injection
  - ***High-level modules should not depend on low-level modules. Both should depend on abstractions.***
  - ***Abstractions should not depend on details. Details should depend on abstractions.***

Coding vs. Programming
Understanding Design Principles

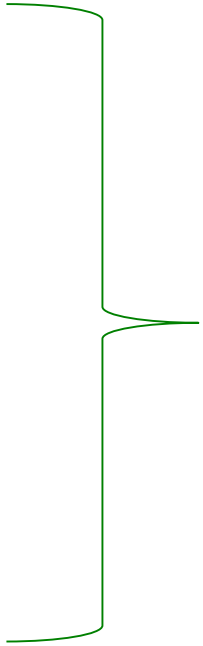
Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion/Injection



"Agile Software Development: Principles, Patterns, and Practices“ by Robert Martin

Understanding Design Principles

Three common characteristics of a BAD design:

Rigidity - It is hard to change because every change affects too many other parts of the system.

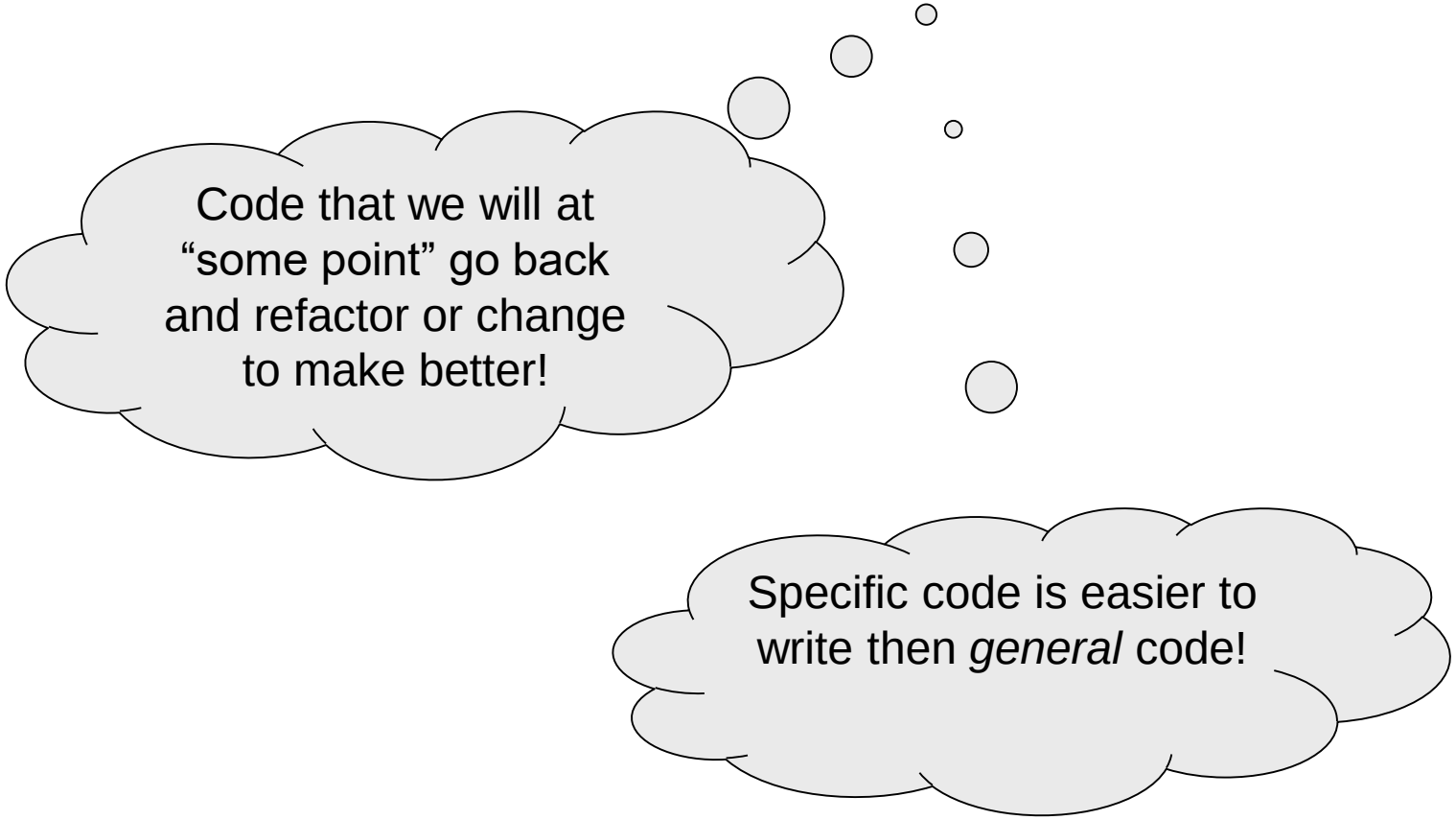
Fragility - When you make a change, unexpected parts of the system break.

Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

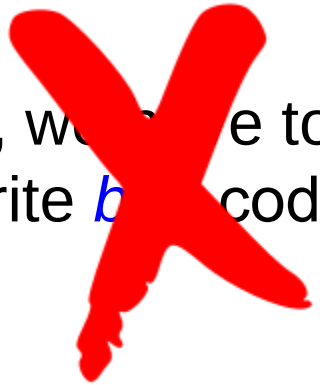
Robert Martin

Abstraction is the Key

In order to go fast, we have to accept that we will write *bad* code…



Abstraction is the Key



In order to go fast, we have to accept that we will write *bad* code…

Abstraction does not mean you have to solve every specific problem!

Abstraction allows you to build an architecture that will not force you to start from scratch every time you need to *pivot* or discover a new requirement.

More pragmatic and economical to build flexible architecture.