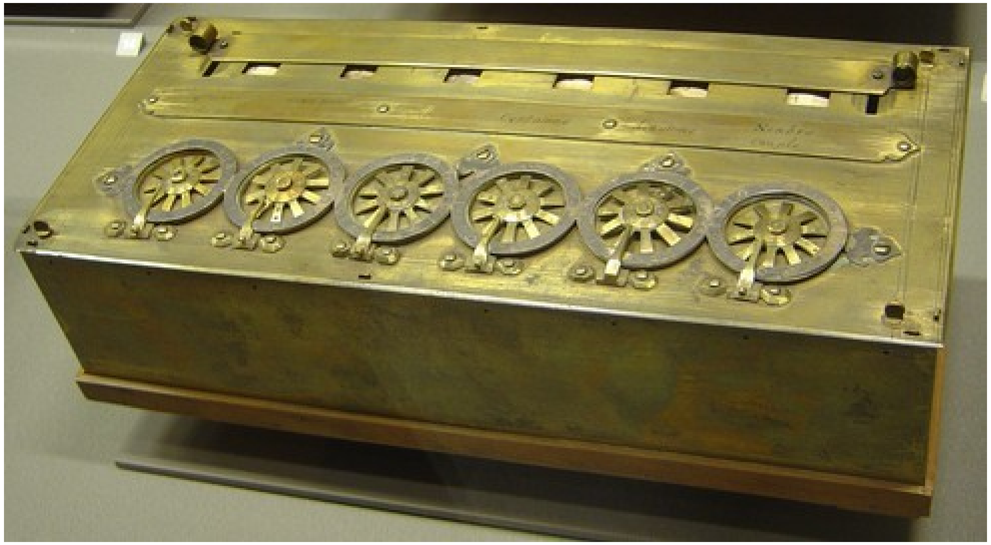
# Essence of Software and SOLID principles

Jean-Luc Delarbre

# Some famous machines

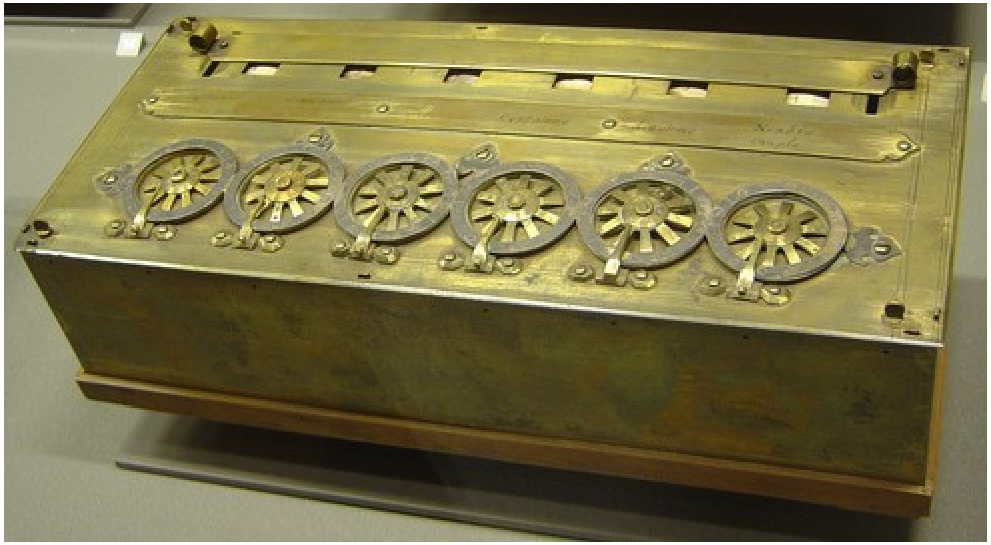


La Pascaline (1652)

First IBM PC (1981)

What is the main difference between those two machines ?

# Some famous machines



La Pascaline (1652)

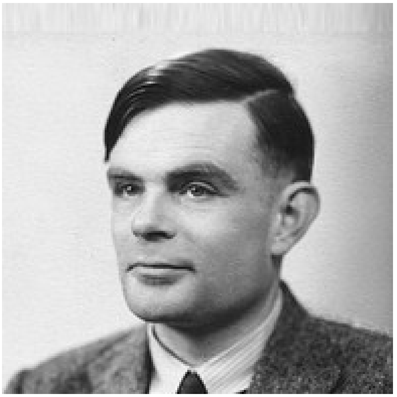
First IBM PC (1981)

What is the main difference between those two machines ?

**NOT THE HARDWARE**

# A little bit of computer science history

- **What makes a computer what it really is, is the work of:**

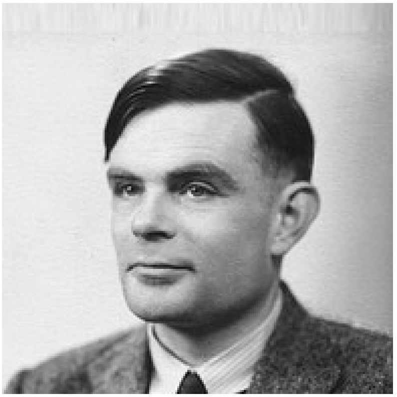


Turing machine

Alan Turing

# A little bit of computer science history

- **What makes a computer what it really is, is the work of:**



Turing machine

Alan Turing

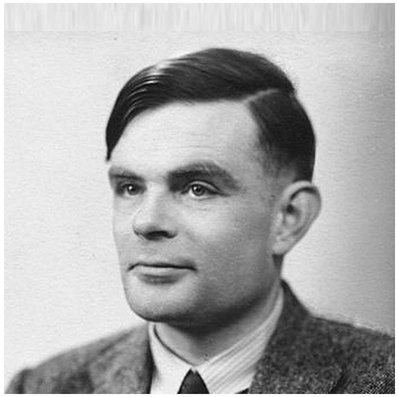
Entscheidungsproblem

David Hilbert

- **Entscheidungsproblem (Decision problem): The Crisis in the Foundations of Mathematics (1928)**
  - Find a machine (algorithm) that answers “Yes” or “No” whether a statement is *universally* valid.
- **Computer (Universal Turing machine) is a side effect of the answer brought to those fundamental questions (1936)**
  - Behavior defined by a program

# A little bit of computer science history

- **What makes a computer what it really is, is the work of:**



Turing machine

Alan Turing

Entscheidungsproblem

David Hilbert

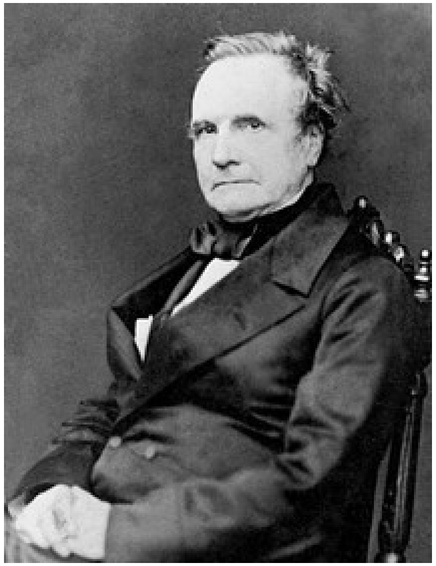
Lambda calculus
Church-Turing thesis

Alonzo Church

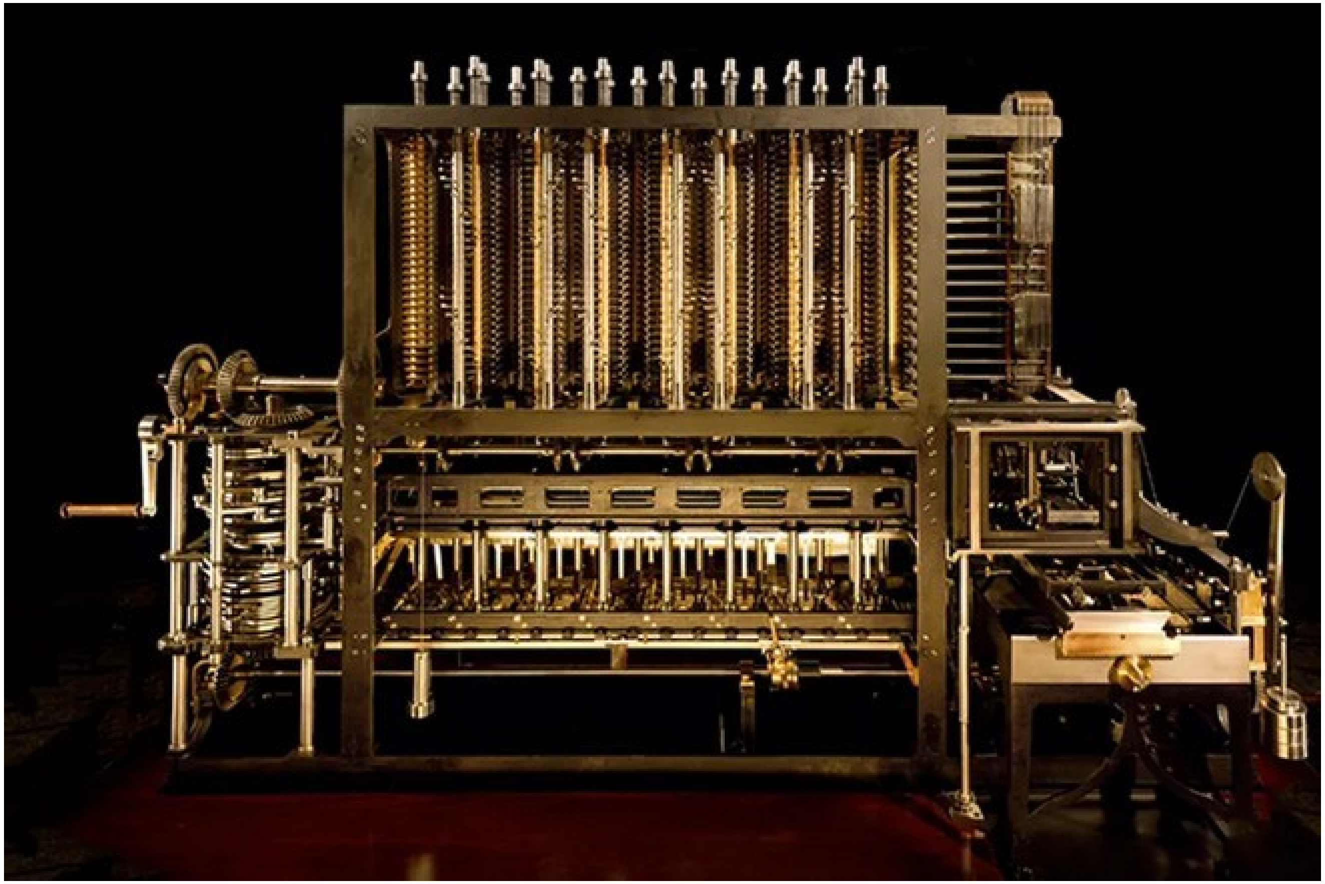
- **Entscheidungsproblem (Decision problem): The Crisis in the Foundations of Mathematics (1928)**
  - Find a machine (algorithm) that answers “Yes” or “No” whether a statement is *universally* valid.
- **Computer (Universal Turing machine) is a side effect of the answer brought to those fundamental questions (1936)**
  - Behavior defined by a program

# The other half of computer history: The engineers work (1/2)

- **Computer design starts before computer science**



Charles Babbage



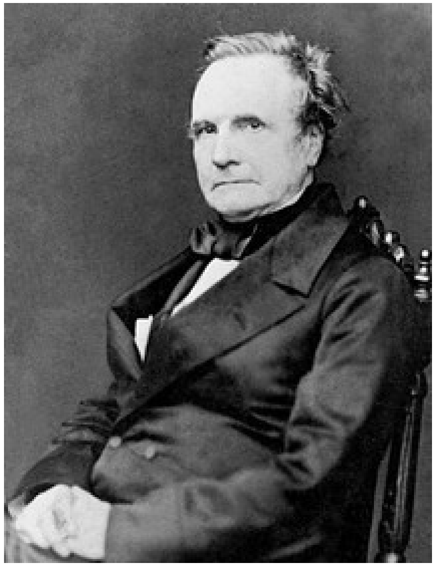
Height: ~ 2,5 m

Difference engine 1822

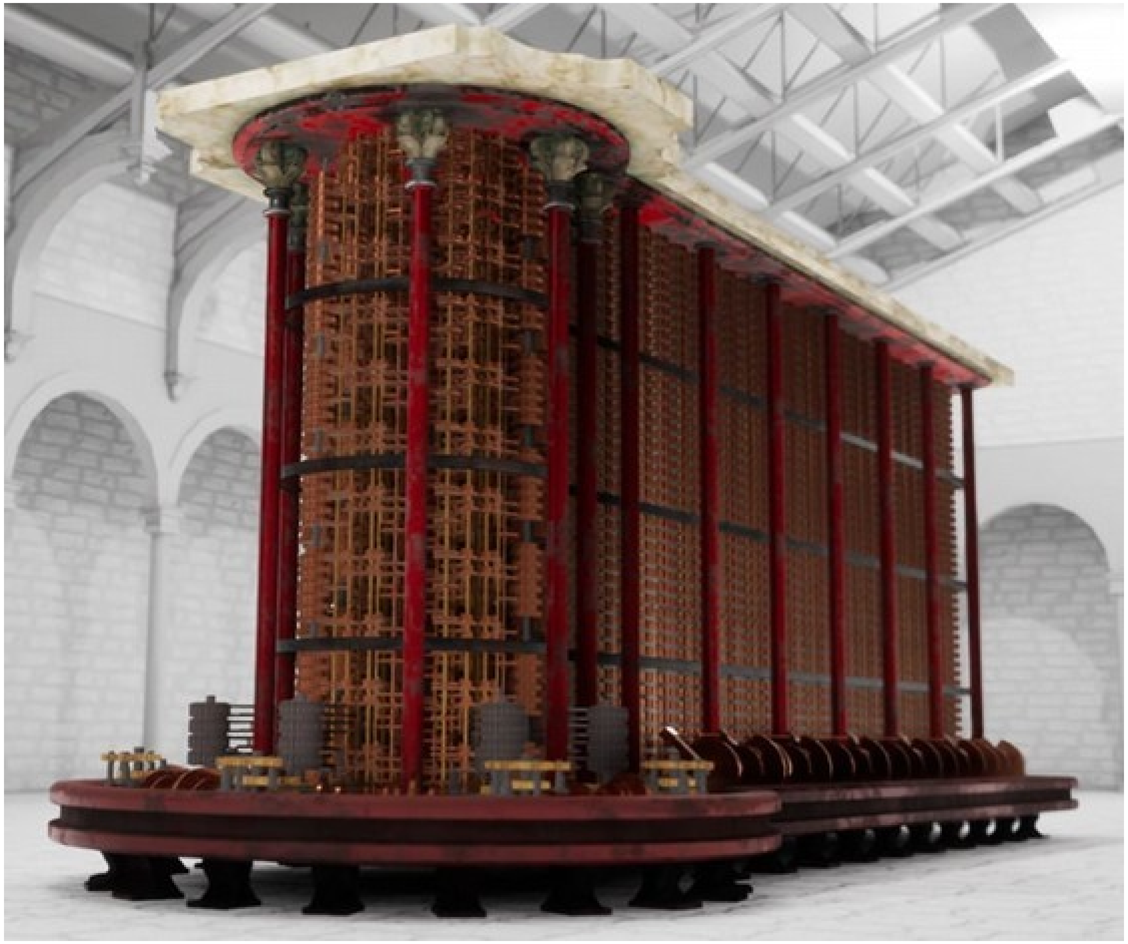
Compute mathematical functions using polynomial interpolation

# The other half of computer history: The engineers work (1/2)

- **Computer design starts before computer science**



Charles Babbage



Height:
~ 3,5 m

Analytical engine (Turing complete) 1837

Capable of *<u>any computation</u>*

# The other half of computer history: The engineers work (2/2)

- **May be not the first programmer but she had glimpses of the future**

Ada Lovelace

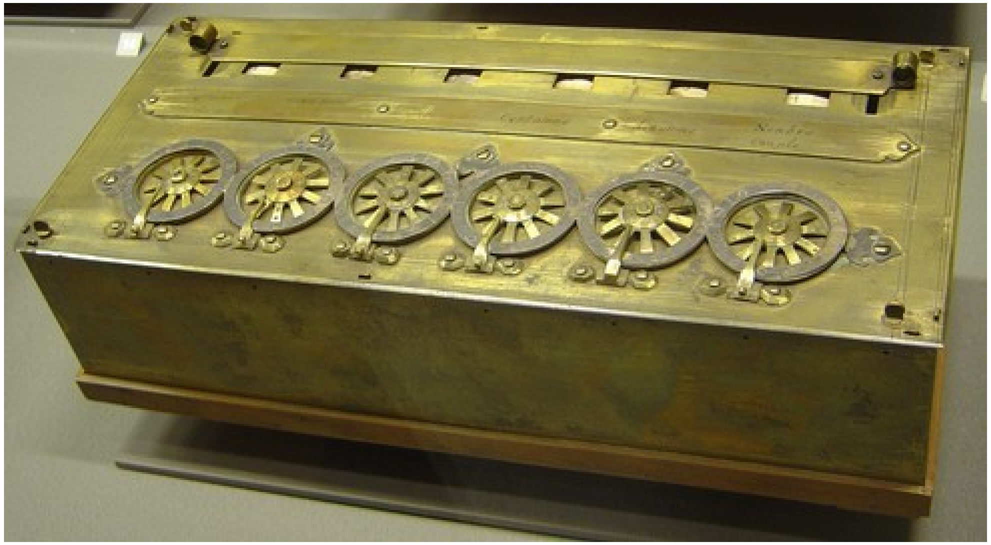
- **More than writing a program for the analytical engine to compute Bernoulli numbers, she was the first to recognize that the machine had applications beyond pure calculation.**

> “The engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.”

> “This example illustrates how the cards are able to reproduce *all the operations* which intellect performs in order to attain a determinate result, if these operations are themselves capable of being precisely defined” [1843]

- **Babbage and Lovelace knew this, but didn’t prove it.**
- **Turing did it in 1936.**

# The fundamental difference



La Pascaline (1652)

First IBM PC (1981)

What is the main difference between those two machines ?

**Programmability**

The behavior of the computer is defined its program, and it is replaceable.

# Essence of software

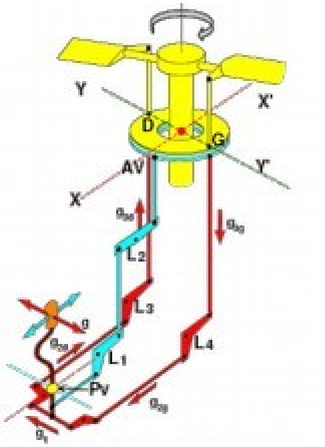
- **Computer are programmed with software**
  - Software is easily replaceable unlike hardware
- **Are we able to modify our code easily?**
  - Do we need to rewrite from scratch for any little changes
- **Our guideline from now**
  - How to make software easily/effectively modifiable?
- **The SOLID principles**

Robert C. Martin
(Uncle Bob)

# Orthogonality

- Which is the easiest tap to use ?

- An aircraft uneasy to pilot



# Single Responsibility Principle

- **A component should have only one reason to change**
  - Responsibility = axis/reason of change
- **Responsibilities (across components) should be independent**
  - No impact of *changes* to one another
  - Orthogonal system
- **Simple concept but hard to get it right**

# Open-Close Principle (1/2)

Bertrand Meyer

- **Easy to change software**
  - Must be designed for it
  - Must be based upon a stable design
    - Contrary of: one change ⇒ cascade of changes
      - Code fragile, rigid, unpredictable and unreusable

- **Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.**
  - Modules that never change
  - Changes done by adding new code
  - ??? How to get changes, if we avoid it

# Open-Close Principle (2/2)

- **Abstraction are the stable elements**
  - Interface in java
- **Extending code done thanks to implementation**
  - Class in java
- **Choose strategic closure (abstraction)**
  - Impossible to be 100% close
  - If new abstraction required
    - Refactor at iso features (check the tests still pass)
    - Then extend for new behavior

# Liskov substitution principle (1/2)

Barbara Liskov

- **Extend is the key of OCP**
  - How to do it well ?

- **FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.**

# Liskov substitution principle (2/2)

- **Design by contract**
  - An API is not only a set of method signatures
    - Method signature
    - Preconditions
    - Postconditions
    - Invariants
- **If the contract is broken OCP no longer works**

# Interface segregation principle (1/2)

- ***Services*** **should not have too many responsibilities (SRP)**

- ***Client*** **should not depend on unnecessary things**

  CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE

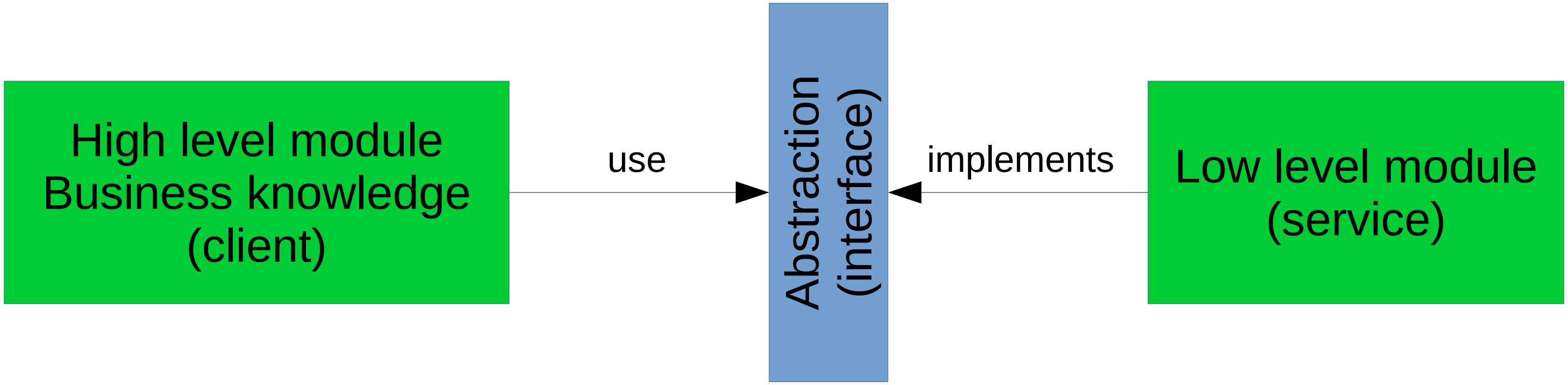
# Interface segregation principle (2/2)

- **Avoid fat interface: split into “role interface” focused and dedicated**
  - Interface pollution
  - Minimize coupling with client
    - If services change, clients are impacted but inverse is true
    - Role interfaces minimize those impacts
  - Ease refactoring (malleable software) of service that implements many role interfaces (shared same data)
    - Information hidding (Parnas 1971)
  - Prefer polyadic form vs monadic

# Dependency inversion principle (1/2)

- A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.
- B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

- **Related to IoC (inversion of control) and DI (Dependency injection)**
- **DIP is the structural implication of OCP and LSP**



# Dependency inversion principle (2/2)

- **Depending on low level modules makes high level modules unreusable**
  - However they holds
    - Higher business value
    - High level policy
    - Strategic decision
  - Changes in low level modules shall not affect high level modules

# Bad design signs

- **Rigidity**
  - Change some part affect many other parts (interdependencies ⇒ high coupling, hard evaluate cost for changes)
- **Fragility**
  - Change some part break the system (not closed to modification hard to be reliable)
- **Immobility**
  - Hard to reuse / extract some part, everything is coupled. Cheaper to redevelop a module

# Software is aimed for changes

- **Software shall do what has been specified**
  - Expected by the client
  - No bugs
- **Computer behavior are defined by programs**
  - Expected to be softly modified
  - From the beginning of software science

# Next objective

- **Design patterns are SOLID**

# Reference

- **SOLID principles**
  - Uncle Bob books
- **The Pragmatic Programmer - Andrew Hunt, David Thomas**
- **Charles Petzold – The Annotated Turing**
- **Thomas Petrachi**
  - https://www.youtube.com/channel/UCSeroUKYavGiiMu-Zhwbg7w
- **Search: Babbage, Difference Machine and Analytical Engine**
  - Computer History Museum (YouTube)
    - Ada Lovelace: The Making of a Computer Scientist
      - https://www.youtube.com/watch?v=2UxjkGePZ48
  - Dartmouth (YouTube)
    - Mission Impossible: Constructing Charles Babbage's Analytical Engine
      - https://www.youtube.com/watch?v=lnFe4UUE4KM