OBJECT-ORIENTED DESIGN & PROGRAMMING

INTRODUCTION TO OBJECT-ORIENTED DESIGN AND PROGRAMMING TECHNOLOGY (OOD/OOP). MAIN PHASES IN OBJECT-ORIENTED DESIGN AND TECHNIQUES IN OBJECT-ORIENTED PROGRAMMING. PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION ISSUES FOR OBJECT-ORIENTED LANGUAGES.

(FROM “CODER” TO “PROGRAMMER”)

GENERAL OVERVIEW

- Concepts of Object-oriented programming
- Review of Java programming
- Concepts and diagrams of UML
- All kinds of design patterns
- Things to make you a better programmer and/or software designer.

EXAMPLES OF UML AND DESIGN PATTERNS

https://www.uml-diagrams.org/index-examples.html

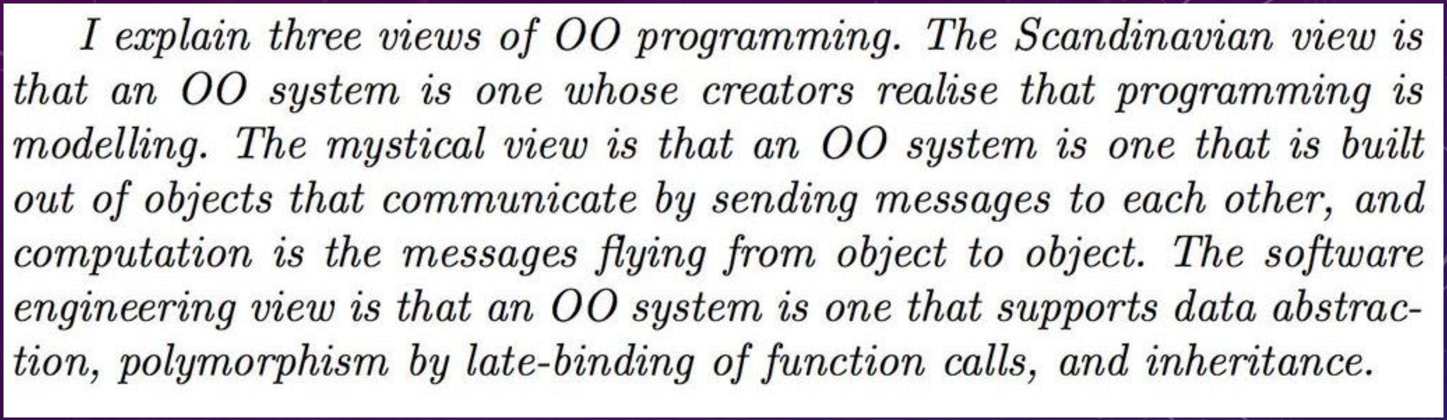
https://dzone.com/refcardz/design-patterns

PHASES OF SOFTWARE DEVELOPMENT

- Specification and analysis
- Design
- Implementation
- Maintenance

- We will focus on the design and implementation phases

QUOTES



I explain three views of OO programming. The Scandinavian view is that an OO system is one whose creators realise that programming is modelling. The mystical view is that an OO system is one that is built out of objects that communicate by sending messages to each other, and computation is the messages flying from object to object. The software engineering view is that an OO system is one that supports data abstraction, polymorphism by late-binding of function calls, and inheritance.

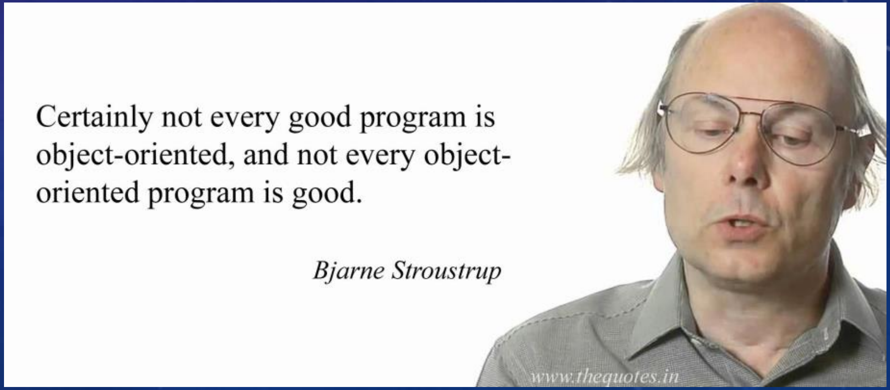
The Myths of Object-Orientation, James Noble.

“Software is too important to be left to programmers”

— Meilir Page-Jones

Never trust a "Software Engineer" who can't code.

— Adam J. Conover



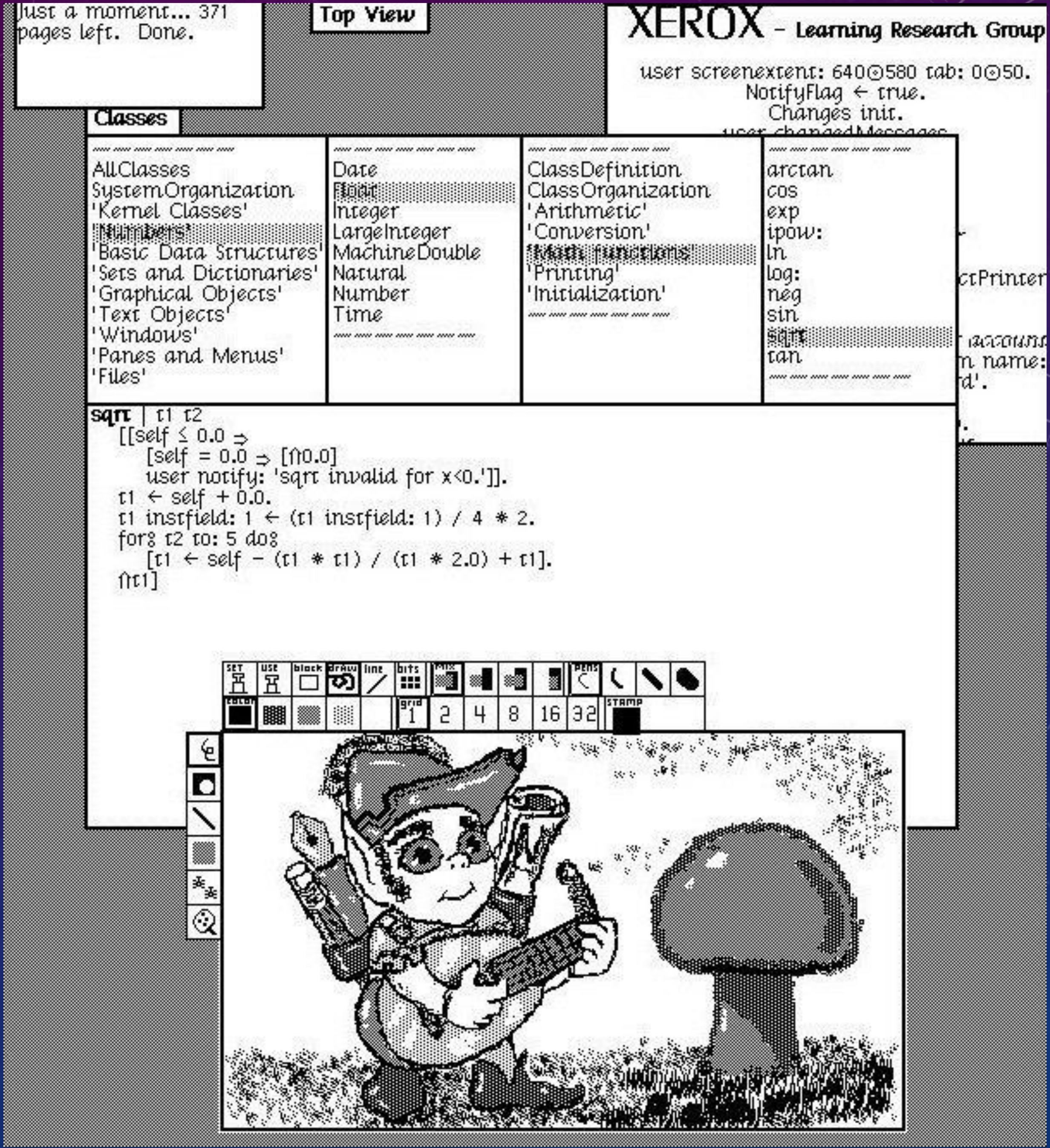
Certainly not every good program is object-oriented, and not every object-oriented program is good.

Bjarne Stroustrup

www.thequotes.in

See: https://www.defprogramming.com/quotes-tagged-with/oop/

The "Mystical" View…



```
Just a moment... 371
pages left.  Done.
```

Top View

XEROX - Learning Research Group

```
user screenextent: 640⊙580 tab: 0⊙50.
NotifyFlag ← true.
Changes init.
```

Classes

AllClasses	Date	ClassDefinition	arctan
SystemOrganization	Float	ClassOrganization	cos
'Kernel Classes'	Integer	'Arithmetic'	exp
'Numbers'	LargeInteger	'Conversion'	ipow:
'Basic Data Structures'	MachineDouble	'Math functions'	ln
'Sets and Dictionaries'	Natural	'Printing'	log:
'Graphical Objects'	Number	'Initialization'	neg
'Text Objects'	Time		sin
'Windows'			sqrt
'Panes and Menus'			tan
'Files'			

```
sqrt | t1 t2
    [[self ≤ 0.0 ⇒
        [self = 0.0 ⇒ [⇑0.0]
        user notify: 'sqrt invalid for x<0.']].
    t1 ← self + 0.0.
    t1 instfield: 1 ← (t1 instfield: 1) / 4 ∗ 2.
    for§ t2 to: 5 do§
        [t1 ← self − (t1 ∗ t1) / (t1 ∗ 2.0) + t1].
    ⇑t1]
```

http://www.righto.com/2017/10/the-xerox-alto-smalltalk-and-rewriting.html

ON THE OTHER HAND... OOP ALLOWS FOR:

- Code reuse
- Unit Testing
- Team programming
- Encapsulation of data
- Allows for data and operations on that data to be tightly coupled.
    - (“Functional Programmers” may argue that this is NOT a “Good Thing!”)
- Development of libraries and frameworksparadigms
- Software maintenance

Note: Other programming paradigms love for all of the above as well!

PROGRAMMING LANGUAGE PARADIGMS

- Imperative (Procedural) Programming (C, Pascal)

  Paradigm primarily based on the assignment of values to variables and data passed to functions/procedures.

- Functional Programming (LISP/Scheme, SML, F#)

  Paradigm based primarily on the use of "High order Functions."

  Programs seen as one big composite function.

- Object-Oriented Programming (C++, Java)

  Paradigm based primarily on the passing of messages

  Modern OOP incorporates elements of both procedureal and functional

- Logic Programming (PROLOG, Erlang)

  Paradigm based primarily on the declaration of needed facts and the use of automated deduction to derive results

THE DEVELOPMENT OF OBJECT-ORIENTED PROGRAMMING

Simula

- First OOP language in the 1960s
- Developed by researchers at the Norwegian Computing Center for use in simulation
- Introduced the foundational concept of “objects”
- Its use as a general programming language was not at first realized

SmallTalk

- Developed in 1970s at Xerox Parc
- Based on idea of objects from Simula
- Called SmallTalk because meant to be easy to use (i.e., easy enough for kids)
- Its use as a general programming language was not at first realized
- The fully further the concepts of OOP into what we know it today
- Still an excellent language to learn OOP with

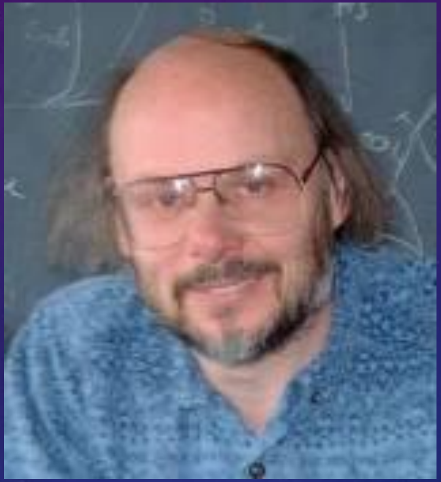
ALAN KAY (DEVELOPED SMALLTALK)

C++

- About the same time that SmallTalk was being developed, C++ was being developed as an OOP version of C at Bell Labs by Bjarne Stroustrup.

- He also based the language on SmallTalk after hearing about it when doing his PhD at Cambridge University.

BJARNE STROUSTRUP (DEVELOPED C++)



Java

- Developed in early 1990s at Sun Microsystems
- Originally developed for use in embedded systems
- Saw its potential for use on the web
- Is a general programming language, not just for web programming

JAMES GOSLING (DEVELOPED JAVA)

USE OF OOP

- First major conference on OOP and OOD held in 1986 – called OOPSLA for:
- "Object-Oriented Programming Systems, Languages, and Applications"
- The OOPSLA conference has since been renamed SPLASH.

OBJECT-ORIENTED PROGRAMMING

Objects are entities which have *attributes* and *behavior*. (This is the same as physical objects in the real world)

State is stored in what is called generically *instance variables*. Behavior is provided by what are called *methods*.

Fundamental Features of Object-Oriented Programming Languages

- Encapsulation
  What is encapsulation?
- Inheritance
  What it inheritance?
- Polymorphism
  What is polymorphism?

Overview of Object-Oriented Analysis and Design

- Object-oriented programming became well known around 1981, when it appeared on the cover of Byte magazine

- The 1980s was the decade of focus on object-oriented programming

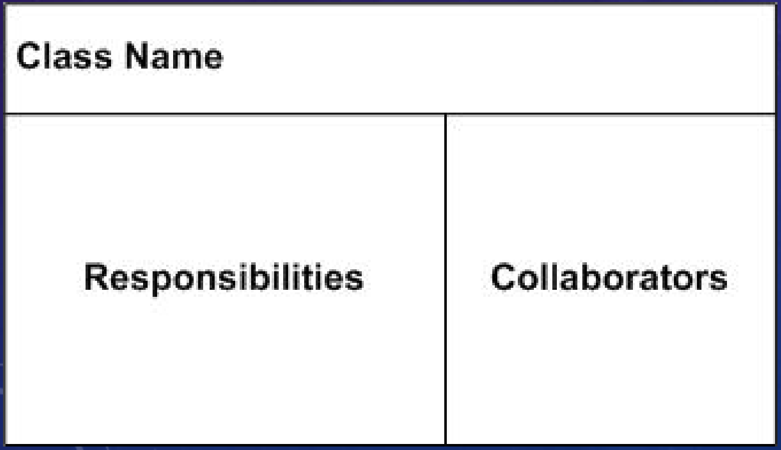
- The 1990s began focus on object-oriented approaches to object-oriented design

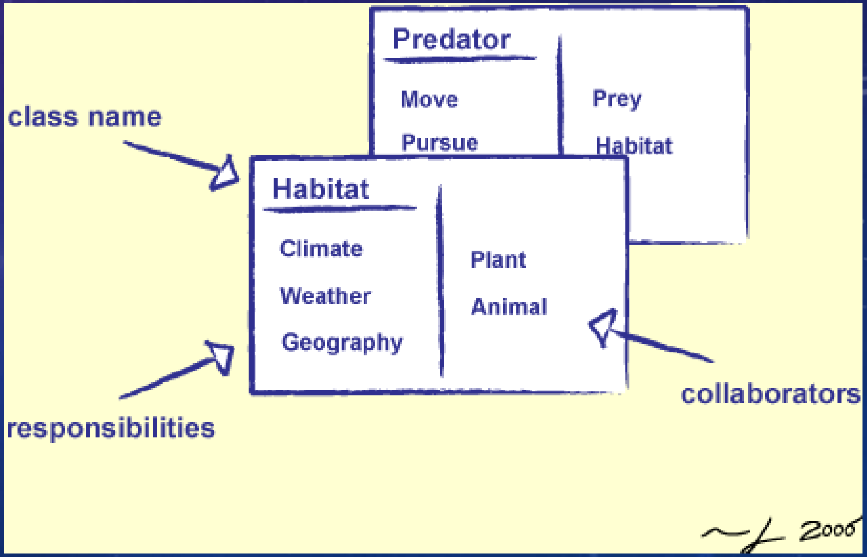
Early Design Methodologies

CRC Cards (1989)

Simple use of index cards, putting one class on each card. Helps limit the size of classes (what can fit on a card) and they associated.

Originally designed as a teaching tool but became more generally used.





Responsibility-Driven Design (RDD) (1990)

Views design as involving roles that entities play, in which each role has a specific set of responsibilities, for example,

information provider
coordinator

Roles are satisfied in implementation by objects, i.e., object implement responsibilities.

Applications as Responsibilities and Collaborations

Application = a set of interacting objects

Object = an implementation of one or more roles

Role = a set of related responsibilities

Responsibility = an obligation to perform a task or know information

Collaboration = an interaction of objects or roles (or both)

Contract = an agreement outlining the terms of a collaboration

User Point-of-View

Application involves information, services, and rules within a given domain.

Designer Point-of-View

Application involves roles (sets of responsibilities) and collaborations.

Implementer Point-of-View

Application involves the implementation of responsibilities
and concern with application-specific objects (e.g., objects related to the user interface).

Roles vs. Objects

Roles and objects are the same when a given role is always played by the same object type.

When more than one kind of object can fulfill the same role, then a role is viewed as a set of responsibilities that can be fulfilled in more than one way.

Cohesion vs. Coupling

Two important aspects of a designed component are referred to as **cohesion** and **coupling**.

Cohesion refers to the degree to which the responsibilities of a given components are a meaningful unit. Greater cohesion reflects better design.

Coupling refers to the degree that one components must access directly access the data of another component. Less coupling reflects better design.

Refactoring

Refactoring is the process of improving the internal structure (both design and code) of software without altering its external behavior.

IDEs (like NetBeans, IntelliJ IDEA, Eclipse, etc.) usually have some built-in refactoring tools.

Other Design Methodologies

(early 1990s)

James Rumbaugh, Ivar Jacobson, and Grady Booch (who created the Rational Company, now a part of IBM) each developed their own, widely known, object-oriented design methodologies.

Would go on to be very influential in OOD.

Object-Oriented Analysis

Object-oriented analysis is a means of deter-mining what aspects of a given problem can be viewed as objects in an eventual object-oriented design.

Thus,

- OOA (object-oriented analysis)
- OOD (object-oriented design)
- OOP (object-oriented programming)

EXAMPLE: FedEx System

For example, suppose we were to develop an overall software system to be used by FedEx.

The business model would involve things like how package shipments are charged, what their costs are, how their profits are determined, etc.

The domain model would identify what entities there are (physical or conceptual) in the system, things like packages, delivery trucks, customer accounts, etc.

Overview of the Unified Modeling Language

(UML)

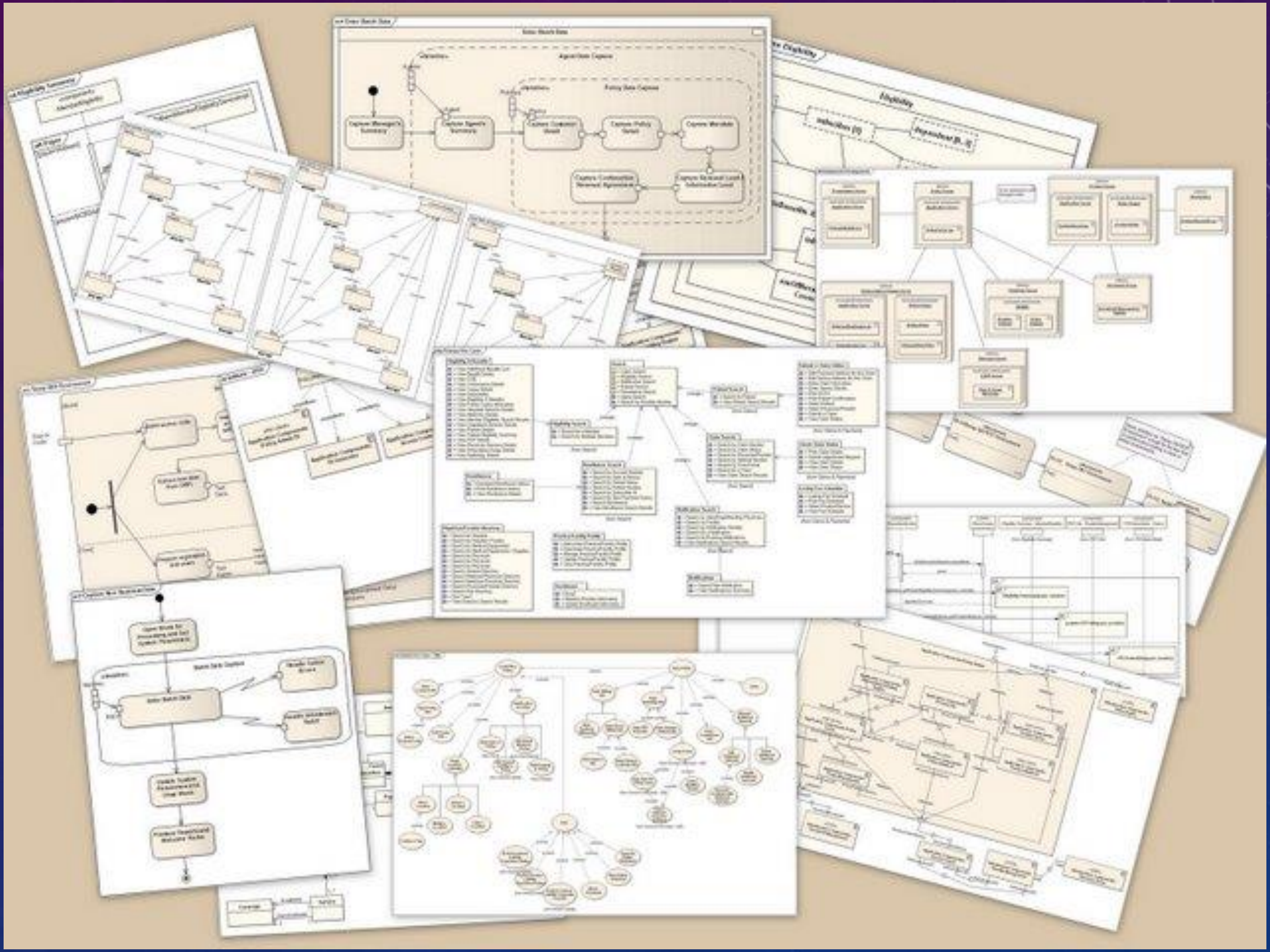
In 1980’s more and more organizations started using OOP.

There was a need for a standardized methodology of OOD.

By early 1990s, different companies using different notations (even within the same company).

A better solution was needed!

UML



Grady Booch, James Rumbaugh, and Ivar Jacobson developing own graphical design languages

In 1994, James Rumbaugh joined Grady Booch at Rational, and soon after, Ivar Jacobson joined them also.

In 1996, the group released an early version of the Unified Modeling Language (UML).

About the same time, and organization called the *Object Management Group (OMG)* invited submissions for a common modeling and specification language.

OMG is a not-for-profit organization issuing guidelines and specifications for object-oriented technologies, including IBM, HP, Microsoft, and Rational Software.

IBM, HP, Microsoft, and Rational formed a group called *UML Partners* in response to the modeling language request from OMG.

UML version 1.1 was developed and submitted to OMG. It was accepted, and in 1997, took responsibility for the ongoing maintenance and revisions of UML.

UML 2.0 released in 2002.

UML specification has been accepted as a ISO (international standard) specification.

UML is the **first specification language to ever become standardized.**

Software Components, Frameworks and Design Patterns

Reuse is an integral aspect of software design.

Software components, frameworks and design patterns represent three forms of reuse.

- **Software Components (reuse of code)**
  Involves the modification and reuse of executable code, just as dragging and modifying button in creating a GUI in visual programming environment. (Java’s version of this is called a JavaBean)

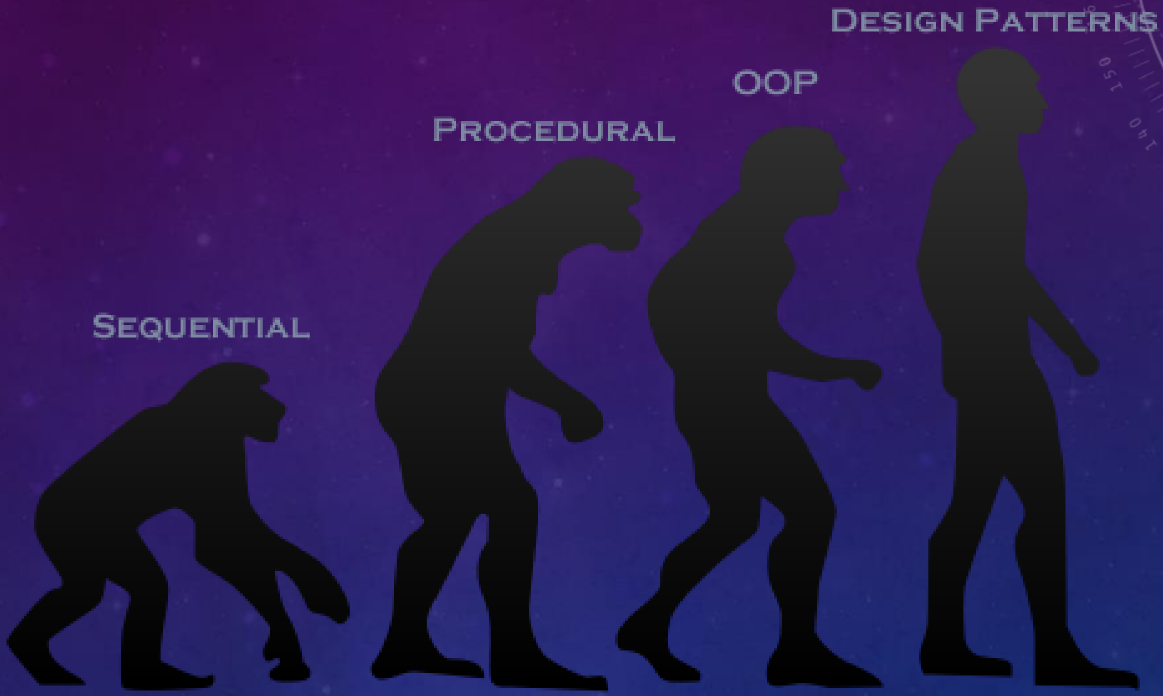
- **Frameworks (reuse of “plug-in” code)**
  A partial implementation that must be completed. Analogous to a motherboard and its hardware components.

- **Design Patterns (reuse of design)**
  A design patterns is a commonly occurring collection of classes and their collaborations that is identified and named, providing for a reusable design approach that others can adopt.

Design Patterns

The concepts of design patterns in software design came from the use of such patterns in architecture.

“Design Patterns: Elements of Reusable Object-Oriented Software” (by Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995)



Programmer Evolution

Example Design Pattern

How to design a class so that only one object instance of that class is created?

Singleton Design Pattern is an elegant solution to this problem.

(Why would we want to guarantee that only ONE instance of any class existed at any given time?)

Object-Oriented Development Methodologies

“Waterfall Model”

- Requirements Analysis
- Specification
- Design
- Implementation
- Testing

Goal is to do each step perfectly before the next step so that a prior step is never repeated. Not a realistic approach.

Agile Development Methodologies

- Changes are expected
- Development approach accommodates changes
- Object-oriented design
- Frequent iterations of design

“Evolving Designs”

This works well in the object-oriented paradigm.

Extreme Programming
(“lightweight” agile development)

- A “lightweight” agile development process
- More focused on coding than on design
- Each iteration very short (few days – few weeks)
- Each iteration focused on
    - *refactoring* (factoring out common code)
    - *enhancements* (new functionality or features)

Key Principles of Extreme Programming

- Simple initial plan that continually gets refined
- Frequent and small releases
- Each release no more than a few weeks apart
- Develop a vocabulary for communicating with client
- Make design as simple as possible.
- Test first – unit test before writing code.
- Refactoring – refactor to continually make system simpler
- Pair programming – write all production code in pairs
- Anyone may change code anywhere in system to improve
- Integrate as soon as task is complete
- Teams should stay fresh and work no more than 40 hours
- Should have customer on site, full time.
- Adopt common standards and conventions for source code, documentation, etc.

THE EXPENSE OF SOFTWARE DEVELOPMENT

COST!

The cost of purchasing, developing, maintaining, and updated software has become the largest expense for many companies.

↓

The aim of the object-oriented software development methodologies is to significantly improve the practice of software development

“ELEGANT” SOFTWARE DESIGN?

USABILITY

IS THE SOFTWARE EASY TO USE?

COMPLETENESS

DOES IT SATISFY ALL THE CLIENT'S NEEDS?

ROBUSTNESS

DOES IT DEAL WITH UNEXPECTED SITUATIONS GRACEFULLY AND AVOID CRASHING?

EFFICIENCY

DOES IT PERFORM THE NECESSARY COMPUTATIONS IN A REASONABLE AMOUNT OF TIME USING A REASONABLE AMOUNT OF MEMORY AND OTHER RESOURCES?

SCALABILITY

WILL IT STILL PERFORM CORRECTLY WHEN THE PROBLEMS GROW IN SIZE BY ORDERS OF MAGNITUDE?

READABILITY

IS IT EASY FOR OTHER TO UNDERSTAND THE SOFTWARE DESIGN AND CODING?

REUSABILITY

CAN THE SOFTWARE BE USED IN ANOTHER COMPLETELY DIFFERENT SETTING?

SIMPLICITY

IS THE DESIGN AND/OR IMPLEMENTATION UNNECESSARILY COMPLEX?

MAINTAINABILITY

CAN DEFECTS BE FOUND AND FIXED EASILY WITHOUT ADDING NEW DEFECTS?

EXTENSIBILITY

CAN THE SOFTWARE BE ENHANCED OR RESTRICTED BY ADDING NEW FEATURES OR REMOVING OLD FEATURES WITHOUT “BREAKING” THE CODE?

WHY WORRY ABOUT SOFTWARE DESIGN?

It is inevitable that large software systems will have bugs.

These bugs can be just annoying, or they can be catastrophic.

EXAMPLES OF SOFTWARE FAILURES

- In 1962, the Mariner I spacecraft was accidentally destroyed.
- In 1980’s, patients were given massive overdoes of radiation.
- In 1990’s, a 9-hour nationwide phone blockage occurred.
- In 2000’s, cash machines dispensed cash freely.

- Radiation machine for treating cancer patients.
- Eventually, bug found after entering and clearing the dose of radiation in a certain sequence – administered an extremely large dose, even though it was clear and corrected in the display.
- Woman lost use of her arm.

- Famous incident in 1990. AT&T long-distance network went down for over 8 hours for the whole US. (cause: missing break statement)

- There was a simple change made to the software.

- The change involved inserting a single break statement in the wrong place.

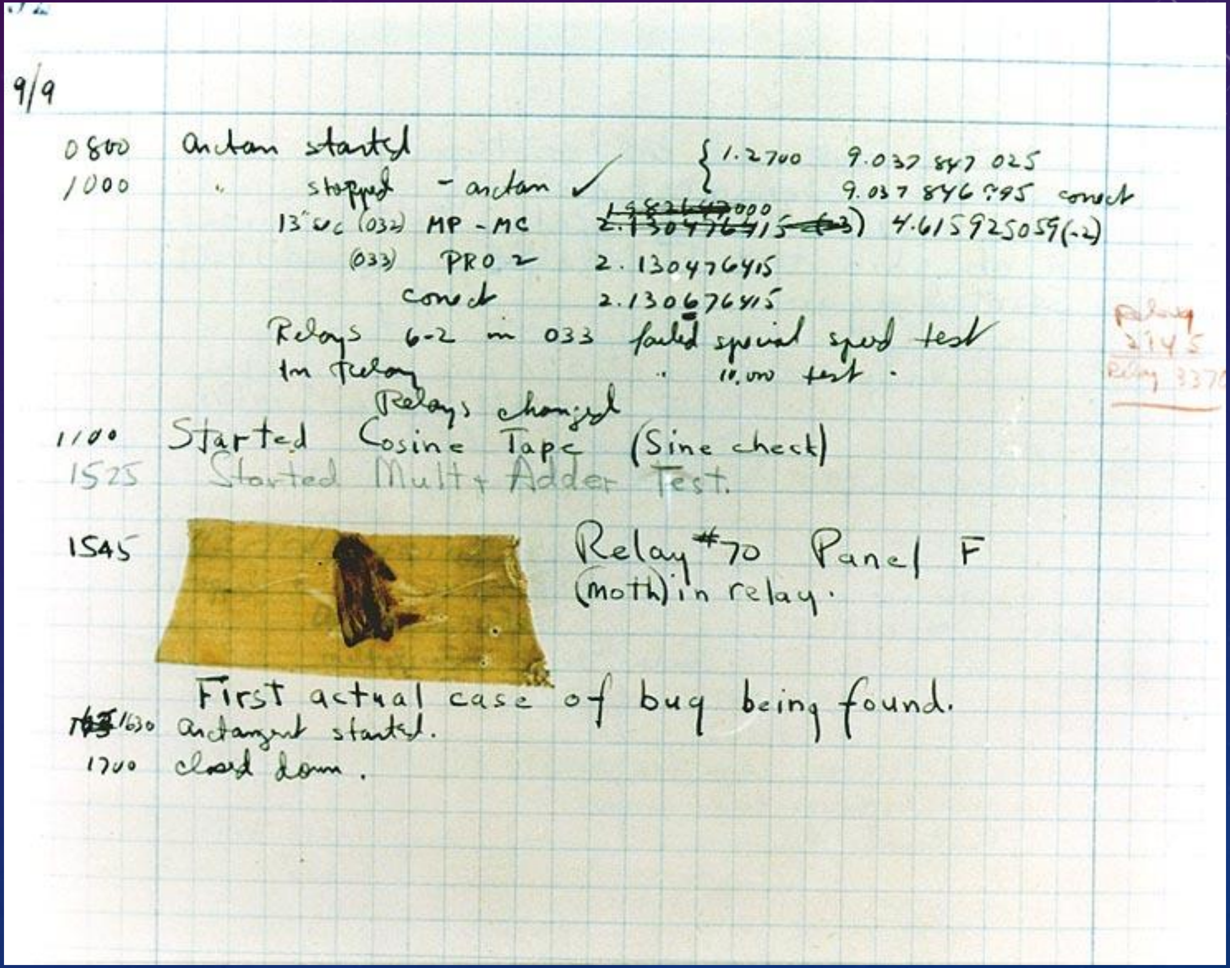
SCALE OF SOFTWARE BUG PROBLEMS

- Software quality issues are estimated to cost the US economy over $2 TRILLION dollars annually!!!
  - https://raygun.com/blog/cost-of-software-errors/

- Software developers spend 80% of their development time finding and fixing bugs

THE FIRST SOFTWARE "BUG"

(SEPTEMBER 9TH, 1945 IN MARK II AT HARVARD)



9/9

0800 Antan started

1000 " stopped - antan ✓ {1.2700 9.037 847 025

9.037 846 995 conct

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)

(033) PRO 2 2.130476415

conct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 10,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Relay 2145

Relay 3370

GOAL OF ELEGANT SOFTWARE DESIGN:

- Minimize the number of initial bugs created
- Maximize the detection and removal of any bugs
- Minimize the number of new bugs added during modification

FORCES WORKING AGAINST BUG-FREE SOFTWARE

- Doing a job right takes time and money, but software developers are under pressure to complete projects quickly

- Writing high quality software requires skill, knowledge and experience that many people do not have

CRITERIA FOR “ELEGANT” SOFTWARE

- Usability
- Completeness
- Robustness
- Efficiency
- Scalability
- Readability
- Reusability
- Simplicity
- Maintainability
- Extensibility